O'REILLY®

TURING 图灵程序设计丛书



Docker开发指南

Using Docker

注重实践、全面实用，让Web应用的开发、测试和部署更简便快捷

[英] Adrian Mouat 著
黄彦邦 译

中国工信出版集团
人民邮电出版社
POSTS & TELECOM PRESS

图灵程序设计丛书

Docker开发指南

Using Docker

Developing and Deploying Software with Containers

[英] Adrian Mouat 著
黄彦邦 译

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo
O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北 京

图书在版编目（ＣＩＰ）数据

Docker开发指南 / (英) 阿德里安·莫阿特
(Adrian Mouat) 著 ; 黄彦邦译. -- 北京 : 人民邮电出
版社, 2017.4
（图灵程序设计丛书）
ISBN 978-7-115-44957-3

Ⅰ. ①D… Ⅱ. ①阿… ②黄… Ⅲ. ①Linux操作系统
—程序设计 Ⅳ. ①TP316.85

中国版本图书馆CIP数据核字(2017)第036596号

内 容 提 要

Docker 容器轻量和可移植的特性尤其适用于动态和分布式的环境，它的兴起给软件开发流程带来了一场革命。本书对 Docker 进行了全面讲解，包括开发、生产以至维护的整个软件生命周期，并对其中可能出现的一些问题进行了探讨，如软件版本差异、开发环境与生产环境的差异、系统安全问题，等等。

本书适合软件开发者、运维工程师和系统管理员，尤其是对 DevOps 模式感兴趣的读者。

◆ 著　　[英] Adrian Mouat
译　　黄彦邦
责任编辑　朱　巍
执行编辑　温　雪
责任印制　彭志环

◆ 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号
邮编　100164　　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　　　　印刷

◆ 开本：700×1000　1/16
印张：18
字数：425千字　　　　2017年 4 月第 1 版
印数：1 – 3 500册　　　　2017年 4 月北京第 1 次印刷

著作权合同登记号　图字：01-2017-0541号

定价：79.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京东工商广字第 8052 号

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 *Make* 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过图书出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

献给那些不计成败、勇于尝试的人们。

目录

第三部分 工具和技术

前言

容器是轻量且可移植的仓库，包含应用程序及其依赖的组件。

诚然，这个描述有点枯燥乏味。不过，若使用恰当，容器对开发流程的改进将会是革命性的。容器使新的软件架构和工作流程成为现实，这股吸引力难以抵挡，仿佛所有大型 IT 公司在一年里都从对 Docker 或容器闻所未闻转为积极研究，甚至已经付诸实践。

Docker 的兴起实在令人惊讶，我印象中没有任何一种技术像它那样，对 IT 产业产生了这么迅速而又深远的影响。我尝试用这本书来帮助你了解容器为什么重要、采用容器技术的好处是什么，以及最重要的是怎样着手使用它。

目标读者

本书试图全面地讲解 Docker，解释使用 Docker 的原因，示范使用方法，以及如何与软件开发流程相结合。本书的内容包括开发、生产以至维护的整个软件生命周期。

本书假设读者只具备 Linux 和软件开发的基础知识，主要的目标读者为软件开发者、运维工程师以及系统管理员（尤其是考虑往 DevOps 方向发展的管理员），而技术型管理者和技术爱好者也能从中受益。

写作目的

Docker 刚迅速兴起的时候，我就有幸知道它，并开始使用。当有机会写作这本书时，我毫不犹豫地抓住了它。假如拙作可以让部分读者了解这场容器化革命并好好地利用它，这将超越我在多年软件开发生涯里所获得的成就。

我真诚希望你会喜欢这本书，希望它能帮助你在公司或组织里使用 Docker。

本书结构

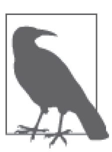
本书大体由以下几部分组成。

- 第一部分首先讲解什么是容器，以及为什么应该关注它。之后将示范 Docker 的基本操作。最后会用较长篇幅来讲解 Docker 的基本概念和技术，其中包括 Docker 命令的概览。
- 第二部分讲解如何将 Docker 应用于软件开发的生命周期。首先讲解如何配置开发环境，然后构建一个简单的 Web 应用，这个 Web 应用的例子将用于整个第二部分。这一部分还会涵盖开发、测试、集成，以及如何部署容器，如何有效地监控和记录生产环境的日志。
- 第三部分的内容更为深入，其中包括在多主机集群环境中，有哪些工具及技巧能使 Docker 容器既安全又可靠地运行。这部分适合已经使用 Docker，并需要了解如何扩展或解决网络和安全问题的读者。

排版约定

本书使用了下述排版约定。

- *楷体*
  表示新术语。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。
- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。
- 斜体等宽字体（*`Constant width italic`*）
  表示应该替换成用户提供的值，或者由上下文决定的值。



该图标表示一般性说明。



该图标表示提示、建议或一般注释。



该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/using-docker/ 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Using Docker* by Adrian Mouat (O'Reilly). Copyright 2016 Adrian Mouat, 978-1-491-91576-9."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920035671.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

致谢

我很感激在我写作这本书的时候，大家给予我的一切帮助、建议和批评。如果下面遗漏了你的名字，请原谅我的疏忽。即便我没有完全按照你们的建议行事，我还是非常感谢你们为我所做的一切。

我要感谢 Ally Hume、Tom Sugden、Lukasz Guminski、Tilaye Alemu、Sebastien Goasguen、Maxim Belooussov、Michael Boelen、Ksenia Burlachenko、Carlos Sanchez、Daniel Bryant、Christoffer Holmstedt、Mike Rathbun、Fabrizio Soppelsa、Yung-Jin Hu、Jouni Miikki 和 Dale Bewley，感谢你们对本书毫无保留的反馈。

针对技术交流和书中一些具体技术的意见，我要感谢 Andrew Kennedy、Peter White、Alex Pollitt、Fintan Ryan、Shaun Crampton、Spike Curtis、Alexis Richardson、Ilya Dmitrichenko、Casey Bisson、Thijs Schnitger、Sheng Liang、Timo Derstappen、Puja Abbassi、Alexander Larsson 和 Kelsey Hightower。同时，还要感谢 Kevin Gaudin 允许我复用 monsterid.js。

感谢 O'Reilly 出版社的工作人员所给予的一切帮助，特别是本书编辑 Brian Anderson 和 Meghan Blanchette。在你们的帮助下，本书的出版工作才得以启动。

还有 Diogo Mónica 和 Mark Coleman，感谢你们在最后关头向我伸出援手。

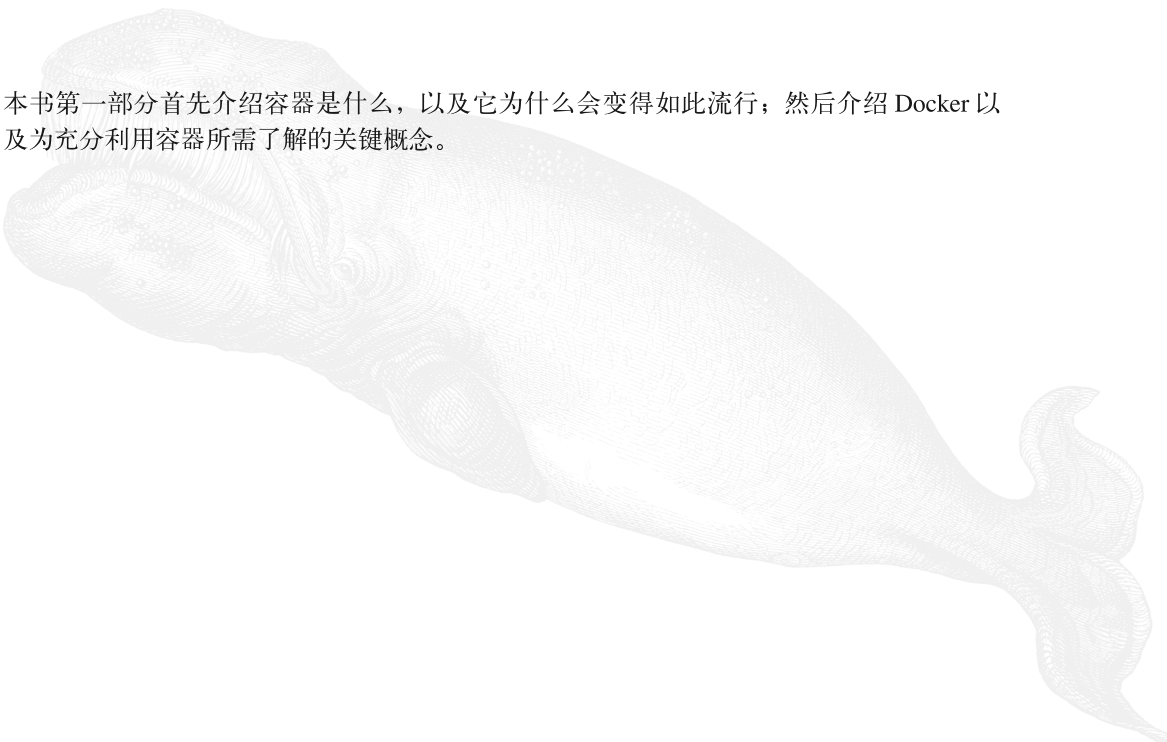
在此我要特别鸣谢两家公司：Container Solutions 和 CloudSoft。Jamie Dobson 和 Container Solutions 给予我很多发表博客文章和在不同活动中演讲的机会，并向我介绍了几位对本书影响深刻的人。CloudSoft 慷慨地让我在写作本书时使用他们的办公室，并主办了爱丁堡 Docker 聚会，这两件事对我而言非常重要。

感谢我的所有朋友和家人，你们容忍我只顾埋头写作，并因为这本书向你们发牢骚。如果你明白我的意思，你应该知道我说的是谁（不过你们不太可能看到这番话）。

最后，感谢 BBC 6 Music 的电台 DJ 提供了原声配乐，包括 Lauren Laverne、Radcliffe and Maconie（Mark Radcliffe 和 Stuart Maconie）、Shaun Keaveny 和 Iggy Pop。

第一部分

背景与基础



本书第一部分首先介绍容器是什么，以及它为什么会变得如此流行；然后介绍 Docker 以及为充分利用容器所需了解的关键概念。

第 1 章

何谓容器，为何需要它

容器从根本上改变了人们开发、发布以及运行软件的方式。软件开发者可以在本地构建软件，因为他们知道软件能够在任何主机环境下运行，无论是 IT 部门的机房、用户的笔记本电脑，还是云端集群，而且运行时并无差异。运维工程师只需专注于维护网络和资源，保证正常运行时间，减少了花在配置环境及解决系统依赖关系上的时间。从规模很小的创业公司，到规模庞大的企业，容器的使用率和理解程度正以惊人的速度提升。可以预见，在未来几年，开发者和运维工程师将以不同形式广泛使用容器技术。

容器是对应用程序及其依赖关系的封装。乍一看容器只是个轻量级的虚拟机，它和虚拟机一样拥有一个被隔离的操作系统实例，用来运行应用程序。

但容器拥有一些优点，使它能实现一些传统虚拟机很难实现甚至无法实现的用例。

- 容器能与主机的操作系统共享资源，因而它的效率高出一个数量级。启动和停止容器均只需一瞬间。相比在主机上直接运行程序，容器的性能损耗非常低，甚至是零损耗。
- 容器具有可移植性，这极有可能彻底解决由于运行环境的些许改变而导致的问题，甚至有可能彻底终止开发者的抱怨：“可是程序在我的计算机上能正常工作！”
- 容器是轻量的，这意味着开发者能同时运行数十个容器，并能模拟分布式系统在真实运行环境下的情况。运维工程师在一台主机上能运行的容器数量,远远超过仅使用虚拟机时。
- 对于最终用户及开发者而言，容器的优势不仅仅体现在云端部署。用户可以下载并执行复杂的应用程序，而无需花费大量时间在配置和安装的问题上，也无需担心对系统本身的改动。另一方面，应用程序的开发者不用再操心用户环境的差异，以及依赖关系是否满足。

更重要的是，虚拟机和容器的根本目标不尽相同。虚拟机的目的是要完整地模拟另一个环境，而容器的目的则是使应用程序能够移植，并把所有依赖关系包含进去。

1.1 容器与虚拟机的比较

虽然容器和虚拟机乍一看很相似，但它们之间有着重大的差异，用图就能轻松解释这些差异。

图 1-1 中有三个应用程序，分别运行在主机的不同虚拟机上。这里需要一个虚拟机管理程序（hypervisor）[1] 来创建及运行虚拟机，控制访问底层操作系统及硬件的权限，以及在必要时解析系统调用接口。每个虚拟机需要一个完整的操作系统、用来运行的应用程序以及所需的程序库。

相比之下，图 1-2 展示了上述三个应用程序在容器化系统中运行的情况。与虚拟机不同，主机的内核 [2] 与容器共享，这意味着容器只能运行与主机一样的内核。应用程序 Y 和 Z 使用相同的程序库，该程序库可以被不同的应用程序同时使用，只需一份，不用复制。容器引擎（container engine）负责启动及停止容器，与虚拟机管理程序差不多。但是，容器中执行的进程与主机自身的进程是等价的，因此没有类似虚拟机管理程序执行所带来的损耗。

虚拟机与容器都可以把主机上的应用程序隔离开来。虚拟技术中的虚拟机管理程序能带来更高一级的隔离性能，是已被公认且千锤百炼的技术。容器技术相对较新，很多公司在取得充分可靠的运行记录前，无法完全信任容器的隔离性能。因此，不难发现有些系统同时采用这两种技术，将容器运行在虚拟机内，这样就能鱼与熊掌兼得。



图 1-1：运行在同一台主机上的 3 个虚拟机

注 1：图中表达的是第二类虚拟机管理程序，如 Virtualbox 和 VMWare Workstation，它运行在操作系统之上。也有第一类虚拟机管理程序，如 Xen，它直接运行在裸机上。

注 2：内核是操作系统的核心部分，负责为应用程序提供关于内存、CPU 以及设备访问的基本系统功能。一个完整的操作系统包括内核和各种系统程序，如启动（init）系统、编译器和窗口管理器。

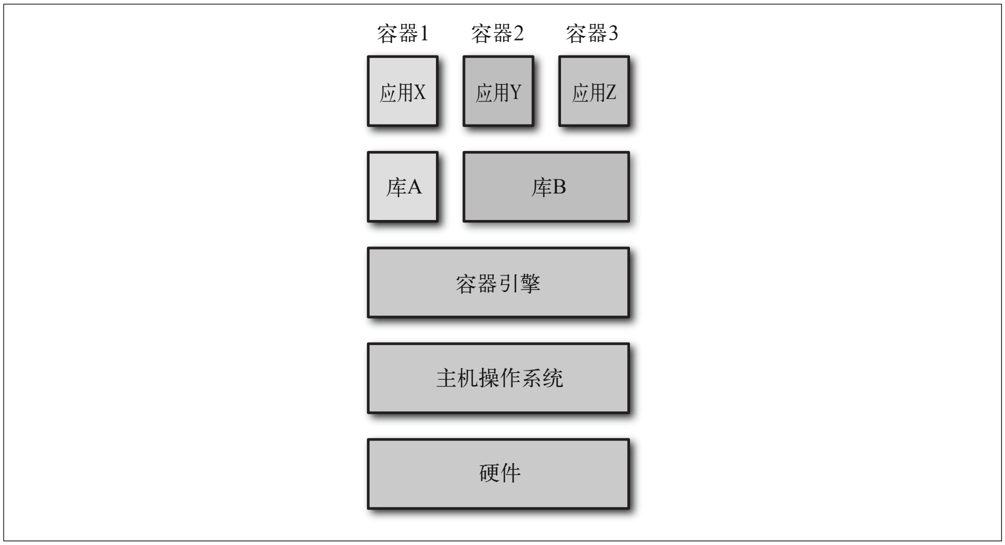


图 1-2：运行在同一台主机上的 3 个容器

1.2 Docker 与容器

容器并不是新概念。几十年来，UNIX 系统一直以 `chroot` 命令来提供简单的文件系统隔离。自 1998 年起，FreeBSD 有了 jail 命令，它把 chroot 的沙盒机制扩展至进程。2001 年左右，Solaris Zones 提供了一个相对完整的容器化技术，但它只能用于 Solaris 操作系统。同样在 2001 年，Parallels 公司（当时称为 SWsoft）推出用于 Linux 的商业容器技术，名为 Virtuozzo，并于 2005 年把核心技术开源，称其为 OpenVZ。[3] 然后，谷歌开始为 Linux 内核开发 CGroups 机制，并开始将它的基础设施容器化。2008 年，Linux 容器（Linux Containers，LXC）项目启动，它把 CGroups、内核命名空间以及 chroot 等技术融合，提供了一套完整的容器方案。最后，Docker 于 2013 年补充了当时容器化技术的不足，将容器技术带入主流。

Docker 利用现有的 Linux 容器技术，以不同方式将其封装及扩展——主要是通过提供可移植的镜像，以及一个用户友好的接口——来创建一套完整的容器创建及发布方案。Docker 平台拥有两个不同部分：负责创建与运行容器的 Docker 引擎，以及用来发布容器的云服务 Docker Hub。

Docker 引擎提供了一个快速且便捷的接口用来运行容器。在此之前，使用如 LXC 等技术运行容器需要相当多的专业知识以及手动操作。Docker Hub 提供大量的公共容器镜像以供下载，方便用户快速上手，并且避免了重复劳动。Docker 还进一步开发了更多的工具，如集群管理工具 Swarm、用于处理容器的图形用户界面 Kitematic，以及部署 Docker 主机的命令行工具 Machine。

由于 Docker 引擎是开源的，这使得 Docker 社区日益壮大，并能借助大家的帮助来修正问

注 3：OpenVZ 一直没有大规模普及，可能是由于需要对内核打补丁所致。

题及改进。Docker 的迅速崛起，使它成为事实上的业界标准，由此业界不得不为容器运行环境及格式发展出一套独立于任何组织的正式标准。2015 年，开放容器促进会（Open Container Initiative，OCI）终于成立，它是一个由 Docker、微软、CoreOS 以及多个重要团体组成的管理架构，目标就是要发展出这一标准。Docker 的容器格式以及运行环境组成了这一工作的基石。

容器技术的兴起很大一部分是由开发者所驱动的，也是他们首次使用了能真正发挥容器潜力的工具。对实施快速迭代开发模式的开发者来说，Docker 容器能迅速启动至关重要，因为他们可以很快看到代码变更后的结果。容器能保障的可移植性及隔离特性，使得开发与运维部门之间更容易协作，因为开发者知道他们的代码在不同环境下都能工作，而运维部门只需专注于容器的托管及服务编排，而无需担心任何关于代码的事。

Docker 为软件开发带来了翻天覆地的变化。假如没有 Docker，在很长一段时间内容器将仍是一种鲜为人知的技术。

航运的比喻

Docker 的哲学经常用航运集装箱的比喻来解释，这或许能解释 Docker 名字的由来。这个比喻大概是这样的。

运输货物时，要用到多种不同的运输**工具**，可能包括货车、叉车、起重机、火车和轮船。这意味着这些工具必须能够处理大小不一、运输需求各异的货物（例如袋装的咖啡、桶装的有毒化学品、盒装的电子产品、成队的豪华轿车、冷冻羊排）。以往这是一道复杂且成本高昂的工序，需要付出大量人力物力。如图 1-3 所示，码头工人在每个中转站手动装卸货物。

联运集装箱的诞生为运输产业带来了一场革命。集装箱的大小有了统一标准，并且设计的出发点是能以最少的人力在不同的运输方式之间搬运。所有的运输机械，无论是叉车和起重机，还是货车、火车和轮船，都为搬运这些集装箱而设计。运输对温度敏感的货物（如食品和药物）时，可以使用具有冷冻及保温功能的集装箱。标准化的好处甚至延伸到其他支撑体系，如集装箱的标签及铅封方式。因此，运输产业只需专注于处理集装箱本身的搬运以及储存的问题，而集装箱内的东西则完全由货物生产商负责。

Docker 的目标是把集装箱的标准化流程运用到 IT 行业中去。近年来，软件系统的多样性激增。在单一机器中运行 LAMP[4] 组合的日子已一去不复返。如今典型的系统可能包含 JavaScript 框架、NoSQL 数据库、消息队列、REST API，以及由各种不同编程语言所写的后端。而这个组合的全部或部分都需要能够在各种不同的硬件上运行——从开发者的笔记本电脑，到公司内部的测试集群，再到云端的生产环境。每个环境都存在差异，它们在不一样的硬件上运行着不一样的操作系统和不同版本的程序库。简而言之，我们的问题与当时运输产业遇到的极为相似——我们正不断付出巨大人力，在不同环境之间移动程序。Docker 容器简化了移动应用程序的工作，好比联运集装箱简

注 4：LAMP 本来是 Linux、Apache、MySQL 和 PHP 的缩写，它们都是常见于 Web 应用的组件。

化了货物运输一样。开发者只需专注于程序开发，再也不用担心测试与正式发布时环境及依赖关系的差异所带来的问题。运维部门则只需专心处理与运行容器相关的核心问题，例如分配资源、启动和停止容器，以及在服务器间的迁移工作。



图 1-3：码头工人在英国布里斯托市工作，摄于 1940 年（来自英国信息部图片处摄影师）

1.3 Docker的历史

2008 年，Solomon Hykes 为了建立一个与编程语言无关的平台即服务（Platform-as-a-Service，PaaS）产品，创立了 dotCloud 公司。编程语言无关这个特性是 dotCloud 独一无二的卖点，因为当时的 PaaS 都必定与某些语言绑定（例如 Heroku 支持 Ruby，Google App Engine 支持 Java 和 Python）。2010 年，dotCloud 参与了 Y Combinator 的加速器计划，认识了一些新的合作伙伴，并开始吸引到一些真正的投资。主要的转机出现在 2013 年 3 月，那时候 dotCloud 将其核心组件 Docker 开源。很多公司都害怕这样做会失去自己那只会下金蛋的鹅，但 dotCloud 则认为把 Docker 转化成一个社区运作的项目会令其受益良多。

Docker 早期的版本只是在简单封装 LXC 以及联合文件系统（union filesystem）之上多加了点东西，但往后无论是发展还是被接受的速度都快得惊人。6 个月内 Docker 就在 GitHub 上获得了 6700 多颗星，以及 175 名非公司员工的贡献者。这导致 dotCloud 把公司名称改为 Docker，并将公司的商业模式重新定位。0.1 版本发布 15 个月后，Docker 1.0 于 2014 年

6 月发布。Docker 1.0 代表着稳定性与可靠性的飞跃——它声称已经“生产就绪”，虽然在这之前就已经有一些公司（如 Spotify 和百度）正式投入使用。同时，Docker 推出了一个名为 Docker Hub 的公共容器仓库，这标志着 Docker 从一个单纯的容器引擎开始转变为一个完整的平台。

其他公司很快就发现了 Docker 的潜力。红帽于 2013 年 9 月成为了它的主要合作伙伴，利用 Docker 来驱动它的 OpenShift 云业务。谷歌、亚马逊和 DigitalOcean 也迅速地在其云服务平台提供 Docker 的支持。有几家创业公司也开始了专门从事 Docker 托管的业务，例如 StackDock。2014 年 10 月，微软公布未来的 Windows Server 将会支持 Docker，这代表着这家让人自然联想到臃肿企业软件的公司，在定位上也作出了重大改变。

在 2014 年 12 月举行的 DockerConEU 上，Docker Swarm 与 Docker Machine 同时面世。Docker Swarm 是一个 Docker 集群管理工具，而 Docker Machine 是个部署 Docker 主机的命令行工具。这表明 Docker 的意图不仅仅是提供 Docker 引擎，而是提供一个完整且综合的容器运行方案。

在同一个月里，CoreOS 宣布开发自家的容器运行环境 rkt 以及 appc 容器规范。2015 年 6 月，DockerCon 在旧金山举行，来自 Docker 的 Solomon Hykes 与来自 CoreOS 的 Alex Polvi 宣布开放容器促进会（当时称为开放容器计划，Open Container Project）正式成立，目的是要发展出一套通用的容器格式与运行环境的标准。

同样是 2015 年 6 月，FreeBSD 项目宣布支持 Docker 利用 ZFS 和 Linux 兼容层来运行。2015 年 8 月，Docker 与微软推出专为 Windows server 开发的 Docker Engine“技术预览”版。

Docker 1.8 版本引入了“内容信任”（content trust）特性，能够核实 Docker 镜像的完整性和发布者身份。对于建立基于 Docker 仓库镜像的可信工作流程，“内容信任”是个很重要的构成部分。

1.4 插件与基础设施

Docker 公司一直勇于承认它的成功有赖于生态系统。当 Docker 公司正全心全意地建造一个稳定且生产就绪的容器引擎时，其他公司（如 CoreOS、WeaveWorks 和 ClusterHQ）则在其他相关领域发展，譬如容器的服务编排和网络连接。但不久之后，显然 Docker 公司正在准备提供一套完整的开箱即用平台，其功能包括联网、储存以及服务编排。为了鼓励生态系统的持续发展，并确保用户能取得各式各样用例的解决方案，Docker 公司宣布将会为 Docker 建立一套模块化且可扩展的框架，原有的组件可以替换为第三方组件，或者加入第三方提供的扩展功能。Docker 公司称这一功能为“包含电池，却可更换”（Batteries Included, But Replaceable），意思是它会提供一套完整的方案，但其中部分可以随意更换。[5]

撰写本书时，插件框架已经可用，但仍在起步阶段。目前已有数个插件用于通过网络来连接容器和数据管理。

注 5：我个人并不喜欢这句话，因为所有电池提供的功能都差不多，而且替换的时候，都必须使用相同大小和电压的电池。我估计这句话是来自 Python 语言的“包含电池”格言，它指的是 Python 本身已经包含一个功能广泛的标准库。

另外，Docker 还遵守一份自己制订的宣言，称为“基础设施构建宣言”（Infrastructure Plumbing Manifesto）。该宣言承诺尽可能重用和改进现有的基础设施组件，并且在社区需要新工具的时候，将可以重复使用的组件回馈给社区。这促使用于运行容器的底层代码被拆分出来，成为 runC 项目，并由 OCI 监管，从而任何人都可以重复利用来搭建其他的容器平台。

1.5 64位Linux

撰写本书时，64 位 Linux 是唯一一个能稳定运行 Docker 且适合用于生产环境的平台。这就意味着，你的计算机必须运行 64 位的 Linux 发行版，而所有的容器也必须是 64 位 Linux。如果你使用的是 Windows 或 Mac OS，你可以在虚拟机里运行 Docker。

其他平台，如 BSD、Solaris 及 Windows Server 的原生支持还处于开发中的不同阶段。由于 Docker 并非使用虚拟化技术，容器必须与主机的内核一致——Windows Server 的容器只能在 Windows Server 的主机上运行，64 位 Linux 的容器只能在 64 位 Linux 的主机上运行。

微服务和单一架构

微服务是容器最主要的用例，也是容器技术兴起的最大推动力。

微服务是一种软件系统开发和构成形式，由小而独立的组件组成，这些组件通过网络互相连接沟通。这与传统的**单一架构**（monolith）软件开发模式相反，后者只有一个庞大的程序，一般由 C++ 或 Java 实现。

当需要扩展一个单一架构的软件时，**纵向扩展**（scale up）往往是唯一选择，也就是说，需要把机器升级，增加内存和使用更强大的 CPU，才能应付更多的负载。相反，微服务则设计成**横向扩展**（scale out），为了满足增长的需求，只需部署多台机器摊分负载即可。微服务架构还可以针对系统中的瓶颈，只扩展某个特定服务所需的资源。但对于单一架构而言，要么扩展所有东西，要么不扩展，而这会造成资源浪费。

就系统复杂度而言，微服务是把双刃剑。每个单独的微服务都应该易于理解和修改，但是，在一个拥有几十到几百个这类服务的系统中，组件之间的交互会导致整体的复杂度增加。

容器与生俱来的轻量级特性及速度，意味着它尤其适合用于微服务架构。与虚拟机相比，容器的体积小很多，并且能快速部署，这使得微服务架构能使用最少的资源，又能迅速应对需求的变化。

更多关于微服务的信息，参见 Sam Newman 的著作《微服务设计》[6]，以及 Martin Fowler 的“微服务资源指南”（Microservice Resource Guide，http://martinfowler.com/microservices/）。

注 6：该书已由人民邮电出版社出版，书号：978-7-115-42026-8。——编者注

第 2 章

安　装

本章概述安装 Docker 的步骤。根据你所用的操作系统，安装时或许会遇到一些小问题；不过运气好的话，安装过程应该是简单和轻松的。如果你已安装 Docker 的最新版本（1.8 或之后），可以直接跳到下一章。

2.1　在 Linux上安装Docker

目前为止，在 Linux 上安装 Docker 最好的方法就是使用 Docker 提供的安装脚本。虽然大部分主流 Linux 发行版都有自己的软件包，但很多时候这些软件包的版本都落后于 Docker 的发布版本。鉴于 Docker 开发的步伐较快，因此绝不能忽略这个问题的严重性。

> **Docker 的系统要求**
>
> Docker 对系统并没有太多要求，不过你需要一个较新的内核（编写本书时是 3.10 或以上版本）。可以通过执行 `uname -r` 来检查你的内核版本。如果你使用的发行版是 RHEL 或 CentOS，便需要 7 或之后的版本。
>
> 还需要注意，系统架构必须是 64 位。系统架构可以通过执行 `uname -m` 查询，结果应为 `x86_64`。

你可以通过 https://get.docker.com 提供的脚本来自动安装 Docker。按照官方的说明，只需执行 `curl -sSL | sh` 或 `wget -qO- | sh` 就可以了，但建议在执行脚本前先检查一下它的内容，确保你接受它对你的系统所作的改动：

```
$ curl https://get.docker.com > /tmp/install.sh
$ cat /tmp/install.sh
...
$ chmod +x /tmp/install.sh
```

```
$ /tmp/install.sh
...
```

这个脚本会先做数个检查，然后用适合你的系统的包安装 Docker。如果它发现系统缺少了一些安全和文件系统功能所需要的依赖关系，还会把它们一并安装。

如果你完全不想使用安装程序，或者希望使用一个安装程序未提供的 Docker 版本，你也可以在 Docker 网站下载二进制文件。这样做的缺点是它不会检查依赖关系，并且以后需要手动安装更新。有关二进制文件的更多信息和下载链接，参见 Docker Binary 网页（https://docs.docker.com/engine/installation/binaries/）。

> **已通过 Docker 1.8 验证**
>
> 撰写本书时，Docker 的版本为 1.8。所有命令皆已通过该版本验证。

2.1.1　将SELinux置于宽容模式下运行

如果你正在运行基于红帽的发行版，包括 RHEL、CentOS 和 Fedora，那么很有可能已经安装了 SELinux 安全模块。

刚开始使用 Docker 时，建议以*宽容*（permissive）模式运行 SELinux，这样 SELinux 将只把错误写进日志，而非强制执行。如果以*强制*（enforcing）模式运行 SELinux，那么很有可能在执行书中的范例时，会遇到各种莫名其妙的“权限不足”（Permission Denied）错误。

要查看你的 SELinux 处于什么模式，可以通过执行 `sestatus` 命令的结果得知。例如：

```
$ sestatus
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   enforcing  ❶
Mode from config file:          error (Success)
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Max kernel policy version:      28
```

❶ 如果这里显示“enforcing”，代表 SELinux 已生效并会强制执行规则。

要将 SELinux 设为宽容模式，只需执行 `sudo setenforce 0`。

更多关于 SELinux 的信息，以及为什么在对 Docker 足够有把握后才应打开 SELinux，参见 13.9.1 节。

2.1.2　不使用sudo命令执行Docker

因为 Docker 运行时需要特殊权限，所以默认执行命令时都必须在前面加上 `sudo`。但这样

做确实使人厌烦，一个可行的解决方法是把用户放进 docker 用户组里。在 Ubuntu 下你可以输入：

```
$ sudo usermod -aG docker $USER
```

如果 docker 用户组不存在，这个命令会创建它，并且把当前的用户添加到组里。然后，你需要先注销并再登入系统。其他 Linux 发行版的做法应该大同小异。

你还需要重启 Docker 服务，不同发行版的操作方法也不一样。Ubuntu 下的操作方法如下：

```
$ sudo service docker restart
```

为简洁起见，书中所有关于 Docker 的命令都会把 sudo 省略。

将用户加入 docker 用户组等同于赋予他 root 权限。因此，你应了解它所带来的安全隐患，如果你的机器是共享的，那么尤其要注意。更多这方面的信息，请参见 Docker security 网页（https://docs.docker.com/engine/security/security/）。

2.2 在Mac OS及Windows上安装Docker

如果你使用的操作系统是 Windows 或 Mac OS，那么你需要某种虚拟化技术才能使用 Docker。[1] 你可以下载整套的虚拟机并按照 Linux 的说明来安装 Docker，或选择安装 Docker Toolbox 工具箱（https://www.docker.com/toolbox）。Docker Toolbox 包含一个极小的 boot2docker 虚拟机，以及本书中将会使用的一些 Docker 工具，例如 Compose 和 Swarm。如果使用 Homebrew 在 Mac 下安装应用，你会发现 boot2docker 的 brew recipe；不过一般我会建议使用官方的 Toolbox 来安装，以免遇到问题。

Toolbox 成功安装后，便可以打开 Docker 的 quickstart 终端使用 Docker。[2] 除此以外，也可以通过以下命令来配置当前的终端：

```
$ docker-machine start default
Starting VM...
Started machines may have new IP addresses. You may need to rerun the
`docker-machine env` command.
$ eval $(docker-machine env default)
```

把你的环境设置妥当，然后就能够访问虚拟机里的 Docker Engine 了。

使用 Docker Toolbox 时务必注意以下事项。

注 1：微软与 Docker 已宣布将联合推动 Windows Server 对 Docker 的支持。未来 Windows Server 用户不再需要透过虚拟化技术便能够启动 Windows 的镜像。（Docker 与微软已于 2016 年 9 月宣布 Windows Server 2016 正式支持 Docker，https://blog.docker.com/2016/09/build-your-first-docker-windows-server-container/。——译者注）

注 2：Docker Toolbox 还包含 Kitematic，一个运行 Docker 容器的图形用户界面。我不会在本书中介绍 Kitematic，但它还是很值得花时间研究的，尤其是刚使用 Docker 的人。

- 本书的范例假设 Docker 运行在主机上。如果你使用 Docker Toolbox，这个假设就不成立了，你需要把提到 `localhost` 的地方一概换成虚拟机的 IP 地址。例如：

```
$ curl localhost:5000
```

需要换成类似下面的例子：

```
$ curl 192.168.59.103:5000
```

虚拟机的 IP 地址可以通过 `docker-machine ip default` 这个命令获得，因此我们还可以稍微做点自动化：

```
$ curl $(docker-machine ip default):5000
```

- 本地操作系统与 Docker 容器之间的映射数据卷必须同时挂载于虚拟机上。Docker Toolbox 在一定程度上可以自动化这个工作，但如果在使用 Docker 数据卷时遇到问题，你得记住这一点。
- 如果你有任何特殊要求，可能需要修改虚拟机的配置，boot2docker 虚拟机里的 /var/lib/boot2docker/profile 文件包含各种不同设定，包括 Docker Engine 的配置。通过修改 /var/lib/boot2docker/bootlocal.sh 文件，你还可以使虚拟机初始化之后执行你自己的脚本。更多详情参见 boot2docker GitHub 仓库（https://github.com/boot2docker/boot2docker）。

如果你跟不上书中的例子，可以执行 `docker-machine ssh default` 来直接登入虚拟机，然后在虚拟机里执行范例的命令。

Docker 的试验性版本

除了常用的稳定版本，Docker 还维护一个试验性版本（experimental build），它包含最新的为测试而设的功能。由于这些功能仍在讨论和开发阶段，在进入稳定版之前它们还有可能会大幅度更改。试验版应只用于正式发布前对新功能的研究，绝不能用于真正的使用环境。

Linux 下可以用这个脚本来安装试验版：

```
$ curl -sSL https://experimental.docker.com/ | sh
```

你也可以从 Docker 网站下载二进制文件。请注意，试验版每天晚上都会更新，网站提供验证码以便确认下载文件是否正确。

2.3 快速确认

可以通过执行 `docker version` 命令得知一切是否已正确安装并且可用。你应该会看到类似下面的输出结果：

```
$ docker version
Client:
 Version:      1.8.1
 API version:  1.20
 Go version:   go1.4.2
 Git commit:   d12ea79
 Built:        Thu Aug 13 02:35:49 UTC 2015
```

```
 OS/Arch:      linux/amd64

Server:
 Version:      1.8.1
 API version:  1.20
 Go version:   go1.4.2
 Git commit:   d12ea79
 Built:        Thu Aug 13 02:35:49 UTC 2015
 OS/Arch:      linux/amd64
```

如果结果相符，这代表你已经准备就绪，可以进入下一章了。但如果你的结果像下面这样：

```
$ docker version
Client:
 Version:      1.8.1
 API version:  1.20
 Go version:   go1.4.2
 Git commit:   d12ea79
 Built:        Thu Aug 13 02:35:49 UTC 2015
 OS/Arch:      linux/amd64
Get http:///var/run/docker.sock/v1.20/version: dial unix /var/run/docker.sock:
no such file or directory.
* Are you trying to connect to a TLS-enabled daemon without TLS?
* Is your docker daemon up and running?
```

这代表 Docker 守护进程并未运行（或客户端无法访问它）。要查出问题所在，可以先执行 `sudo docker daemon` 来手动启动 Docker 守护进程。它或许能给出一些信息，指出哪一部分出错，而那些信息也有助于搜寻答案。（请注意，这个建议只适用于 Linux 主机。如果你使用的是 Docker Toolbox 或类似的工具，请求助于相关文档。）

第 3 章

迈出第一步

在这一章你将迈出使用 Docker 的第一步。本章先从启动和使用一些简单的容器开始，感受一下 Docker 是如何工作的；然后探讨 Dockerfile——构建 Docker 容器的基石，以及为发布容器提供支持的 Docker Registries（寄存服务），最后讲解怎样通过容器利用持久化存储来托管一个键值存储（key-value store）。

3.1 运行第一个镜像

尝试执行下述命令，测试 Docker 的安装是否正确。

```
$ docker run debian echo "Hello World"
```

运行这个命令需要花点时间，实际所需时间依网络速度而定，但最终你应看到类似下面的结果。

```
Unable to find image 'debian' locally
debian:latest: The image you are pulling has been verified
511136ea3c5a: Pull complete
638fd9704285: Pull complete
61f7f4f722fb: Pull complete
Status: Downloaded newer image for debian:latest
Hello World
```

究竟发生了什么？刚才我们调用了 `docker run` 命令，它的功能是负责启动容器。其中的 `debian` 参数是我们打算使用的镜像[1]名称，这里所指的是一个被精简过的 Debian Linux[2] 发

注 1：后面会有镜像更详细的定义，现在暂且把它当成是容器的“模板”。

注 2：官方正式名称为 Debian GNU/Linux。——译者注

行版。输出结果的第一行告诉我们本地没有 Debian 镜像。Docker 便会在 Docker Hub 进行在线搜索，并下载 Debian 最新版本的镜像。镜像下载后，Docker 会将它转成容器并运行，然后在容器中执行我们指定的命令——`echo "Hello World"`。命令的结果则显示在输出内容的最后一行。

如果再次执行同一命令，那就无需再下载镜像了，容器会立即启动。而整个命令的运行时间大概只需一秒，要是想象一下它背后所做的一切事情，就会觉得这个速度十分惊人：Docker 部署并启动了我们的容器，执行我们指定的 `echo` 命令，最后把容器关掉。类似的工作如果用传统的虚拟机执行，所需时间将会是好几秒，甚至是好几分钟。

我们可以用以下命令，请求 Docker 提供一个容器中的 shell。

```
$ docker run -i -t debian /bin/bash
root@622ac5689680:/# echo "Hello from Container-land!"
Hello from Container-land!
root@622ac5689680:/# exit
exit
```

这样你就可以进入容器中的命令行了，和使用 `ssh` 进入远程机器很相似。当中的 `-i` 和 `-t` 参数表示我们想要一个附有 tty 的交互会话（interactive session），`/bin/bash` 参数表示你想获得一个 bash shell。当你退出 shell 时，容器就会停止——主进程运行多久，容器就运行多久。

3.2 基本命令

为了进一步了解 Docker，我们可以启动一个容器，执行不同命令和行动，并观察容器会作出什么反应。首先，启动一个新的容器，不过这次用 `-h` 参数来设定一个新的主机名（hostname）。

```
$ docker run -h CONTAINER -i -t debian /bin/bash
root@CONTAINER:/#
```

如果故意把容器弄坏会怎样呢？

```
root@CONTAINER:/# mv /bin /basket
root@CONTAINER:/# ls
bash: ls: command not found
```

我们移动了 /bin 目录的位置，现在这个容器已经没用了，至少暂时是这样的。[3] 在删掉这个容器之前，先来看看 `ps`、`inspect` 和 `diff` 会显示什么结果。现在，打开一个新的终端（同时保持原来的容器运行），并在主机执行 `docker ps`。运行结果如下：

```
CONTAINER ID  IMAGE  COMMAND        ...     NAMES
00723499fdbf  debian "/bin/bash"    ...     stupefied_turing
```

上面的内容告诉我们目前运行中的容器的一些详细信息。其中大部分信息应无需多作解

注 3：我在演示的时候一般不用 `mv` 而用 `rm`，但我担心有人会不小心在自己的主机上运行 `rm`，因此这里我还是决定使用 `mv`。

释，不过，有一点要特别注意，那就是 Docker 为容器起了一个容易看懂的名称，这里叫作“stupefied_turing”[4],在本机上可以用它来识别这个容器。我们可以把容器的名称或 ID 作为 docker inspect 命令的参数，来获取更多有关某个容器的信息：

```
$ docker inspect stupefied_turing
[
{
    "Id":"00723499fdbfe55c14565dc53d61452519deac72e18a8a6fd7b371ccb75f1d91",
    "Created": "2015-09-14T09:47:20.2064793Z",
    "Path": "/bin/bash",
    "Args": [],
    "State": {
        "Running": true,
...
```

这里有很多有用的信息，但却不太容易读懂。我们可以用 grep 或 --format 参数（需要一个 Go 模板[5]）来过滤感兴趣的信息。例如：

```
$ docker inspect stupefied_turing | grep IPAddress
        "IPAddress": "172.17.0.4",
        "SecondaryIPAddresses": null,
$ docker inspect --format {{.NetworkSettings.IPAddress}} stupefied_turing
172.17.0.4
```

这两行命令都能找出容器的 IP 地址。现在来看看另一个命令，docker diff：

```
$ docker diff stupefied_turing
C /.wh..wh.plnk
A /.wh..wh.plnk/101.715484
D /bin
A /basket
A /basket/bash
A /basket/cat
A /basket/chacl
A /basket/chgrp
A /basket/chmod
...
```

可以看到，在这个运行中的容器内，有哪些文件被改动过；而这个例子中，被删除的文件有 /bin，有新增的 /basket 以及它底下的文件，还有一些存储驱动的相关文件。Docker 容器使用联合文件系统（union file system，UFS），它允许多个文件系统以层级方式挂载，并表现为一个单一的文件系统。镜像的文件系统以只读方式挂载，任何对运行中容器的改变则只会发生在它之上的可读写层。因此，Docker 只需查看最上面的可读写层，便可找出曾对运行系统所作的所有改变。

注 4：Docker 为容器生成的名称，都是由一个随机的形容词，加上一个著名的科学家、工程师或黑客的名字所组成。除了自动生成，也可以用 --name 参数来指定名称（例如 docker run --name boris debian echo "Boo"）。

注 5：这里指的是 Go 编程语言的模板引擎。它是一个功能完整的模板引擎，为过滤和筛选数据提供了很强大的灵活性和能力。在 Docker 网站可以找到更多关于怎样使用 inspect 的信息（https://docs.docker.com/reference/commandline/inspect/）。

关于这个容器的实验即将结束，最后要为大家介绍的是 `docker logs`。以容器名称作为它的参数，就能得知这个容器里曾经发生过的一切事情：

```
$ docker logs stupefied_turing
root@CONTRAINER:/# mv /bin /basket
root@CONTRAINER:/# ls
bash: ls: command not found
```

现在，对这个坏掉了的容器的实验到此结束，是时候把它删掉了。首先，从 shell 退出：

```
root@CONTRAINER:/# exit
exit
$
```

由于 shell 是唯一一个在容器中运行的进程，容器也会同时停止。这时候如果执行 `docker ps`，你会发现已经没有任何正在运行的容器了。

不过，这并非事实的全部。如果键入 `docker ps -a`，它会列出所有容器，包括已经停止的容器（官方说法是“已退出容器”，exited container）。已退出的容器可以用 `docker start` 重启（因为我们已经把之前那个容器的路径弄坏了，所以它已无法重启）。要把容器删掉，需要使用 `docker rm` 命令：

```
$ docker rm stupefied_turing
stupefied_turing
```

清理已停止的容器

如果想删除所有已停止的容器，可以利用 `docker ps -aq -f status=exited` 的结果，结果中包含所有已停止容器的 ID。例如：

```
$ docker rm -v $(docker ps -aq -f status=exited)
```

这个命令很常用，因此你可以考虑把它写成一个 shell 脚本或定义为 alias。请注意 `-v` 参数在这里的作用，它意味着当所有由 Docker 管理的数据卷已经没有和任何容器关联时，都会一律删除。

为了避免已停止的容器的数量不断增加，可以在执行 `docker run` 的时候加上 `--rm` 参数，它的作用是当容器退出时，容器和相关的文件系统会被一并删掉。

好了，现在来看怎样创建一个全新且具有实用价值的容器，一个我们会真正保留下来的容器。[6] 我们打算将一个名为 cowsay 的应用“Docker 化”。如果你不知道什么是 cowsay 的话，待会儿别被吓倒了。首先启动一个容器，并安装一些包：

```
$ docker run -it --name cowsay --hostname cowsay debian bash
root@cowsay:/# apt-get update
...
Reading package lists... Done
root@cowsay:/# apt-get install -y cowsay fortune
```

注 6：虽然我说有实用价值，但严格来说还不至于太有用。

```
...
root@cowsay:/#
```

现在就来试一试吧！

```
root@cowsay:/# /usr/games/fortune | /usr/games/cowsay
 _____________________________________
/ Writing is easy; all you do is sit  \
| staring at the blank sheet of paper |
| until drops of blood form on your   |
| forehead.                           |
|                                     |
\ -- Gene Fowler                      /
 -------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

非常好，一切顺利，把容器留下吧。[7] 要把它转成镜像的话，执行 `docker commit` 即可，无论容器是在运行中还是在停止状态都可以。执行 `docker commit` 时，你需要提供的参数包括容器的名称（“cowsay”）、新镜像的名称（“cowsayimage”），以及用来存放镜像的仓库名称（“test”）：

```
root@cowsay:/# exit
exit
$ docker commit cowsay test/cowsayimage
d1795abbc71e14db39d24628ab335c58b0b45458060d1973af7acf113a0ce61d
```

命令的返回值是这个镜像的唯一识别码（unique ID）。现在，我们创建了一个能随时使用，并且已安装 cowsay 的镜像：

```
$ docker run test/cowsayimage /usr/games/cowsay "Moo"
 _____
< Moo >
 -----
       \   ^__^
        \  (oo)\_______
           (__)\       )\/\
               ||----w |
               ||     ||
```

太棒了！不过其实还有一些问题有待解决。如果需要更改一些东西，那么我们还得手动重复刚才的步骤。假设我们希望基于另一个镜像来创建我们的容器的话，也必须从头开始。更重要的是，这样做的*可重复性*（repeatable）很差；创建镜像的步骤很难与他人分享，把步骤重做也不是件易事，而且容易出错。解决这些问题的方法就是利用 Dockerfile，使创建镜像的过程全部自动化。

注 7：虽然这个容器只能用来玩玩而已，不过你按我所说的来做就好。

3.3 通过Dockerfile创建镜像

简而言之，Dockerfile 是一个描述如何创建 Docker 镜像所需步骤的文本文件。在下面的例子中，我们先创建一个文件夹和文件：

```
$ mkdir cowsay
$ cd cowsay
$ touch Dockerfile
```

然后把下面的内容放入 Dockerfile：

```
FROM debian:wheezy

RUN apt-get update && apt-get install -y cowsay fortune
```

`FROM` 指令指定初始镜像（和之前一样，这里使用 `debian`，但这次我们还指定使用“wheezy”版本）。所有 Dockerfile 一定要有 `FROM` 指令作为第一个非注释指令。`RUN` 指令指定的 shell 命令，是将要在镜像里执行的。这个例子中，它所做的事情其实和之前一样，就是安装 cowsay 和 fortune。

现在已准备好生成镜像了，在同一目录下执行 `docker build` 命令：

```
$ ls
Dockerfile
$ docker build -t test/cowsay-dockerfile .
Sending build context to Docker daemon 2.048 kB
Step 0 : FROM debian:wheezy
 ---> f6fab3b798be
Step 1 : RUN apt-get update && apt-get install -y cowsay fortune
 ---> Running in 29c7bd4b0adc
...
Setting up cowsay (3.03+dfsg1-4) ...
 ---> dd66dc5a99bd
Removing intermediate container 29c7bd4b0adc
Successfully built dd66dc5a99bd
```

创建成功后，就可以与之前一样运行镜像了：

```
$ docker run test/cowsay-dockerfile /usr/games/cowsay "Moo"
```

镜像、容器和联合文件系统

为了使大家明白镜像与容器之间的关系，我必须为大家讲解清楚一个 Docker 使用的核心技术，那就是**联合文件系统**（有时也称为“**联合挂载**”）。联合文件系统允许多个文件系统叠加，并表现为一个单一的文件系统。文件夹中的文件可以来自多个文件系统，但如果有两个文件的路径完全相同，最后挂载的文件则会覆盖较早前挂载的文件。Docker 支持多种不同的联合文件系统实现，包括 AUFS、Overlay、devicemapper、BTRFS 及 ZFS。具体使用哪种实现取决于你所用的系统，可以通过 `docker info` 命令，查看输出结果中“Storage Driver”的值得知。文件系统是可以变更的，但你必须清楚你所做的事，以及它所带来的好处和坏处，否则我不建议你这样做。

Docker 的镜像由多个不同的“**层**”（layer）组成，每一个层都是一个只读的文件系统。Dockerfile 里的每个指令都会创建一个新的层，而这个层将位于前一个层之上。当一个镜像被转化成一个**容器**时（譬如通过 `docker run` 或 `docker create` 命令），Docker 引擎会在镜像之上添加一个处于最上层的可读写文件系统（同时还会对一些配置进行初始化，如 IP 地址、名称、ID，以及资源使用限制等）。

由于不必要的层会使镜像变得臃肿（而且 AUFS 最多只能有 127 个层），你会发现很多 Dockerfile 都把多个 UNIX 命令放在同一个 RUN 指令中，以减少层的数量。

容器可以处于以下几种状态之一：**已创建**（created）、**重启中**（restarting）、**运行中**（running）、**已暂停**（paused）和**已退出**（exited）。“已创建”指容器已通过 `docker create` 命令初始化，但未曾启动。很多时候“已退出”也称为“已停止”，指容器中没有正在运行的进程（虽然“已创建”状态的容器也没有正在运行的进程，但“已退出”的容器至少启动过一次）。容器的主进程退出时，容器也会退出。“已退出”的容器可以用 `docker start` 命令重启。已停止的容器不等于一个镜像，因为前者还会保留对配置、元数据和文件系统的改动。“重启中”状态实际上很少遇见，当 Docker 引擎尝试重启一个启动失败的容器时，它才会出现。

通过利用 Dockerfile 的 `ENTRYPOINT` 指令，我们可以让用户更易于使用这个镜像。`ENTRYPOINT` 指令让我们指定一个可执行文件，同时还能处理传给 `docker run` 的参数。

在 Dockerfile 的最后加上下面这一行：

```
ENTRYPOINT ["/usr/games/cowsay"]
```

现在再次生成新镜像，以后使用这个新镜像时再也不需要指定 cowsay 命令了：

```
$ docker build -t test/cowsay-dockerfile .
...
$ docker run test/cowsay-dockerfile "Moo"
...
```

这样确实简单多了。可是，我们却失去了将容器中 fortune 命令的输出作为 cowsay 输入的能力。为了解决这个问题，我们可以把 `ENTRYPOINT` 指定为一个我们自己的脚本，这种做法在创建 Dockerfile 时是很常见的。现在创建一个新文件 entrypoint.sh，把下面的内容放入文件中，并保存至 Dockerfile 的同一目录下：[8]

```
#!/bin/bash
if [ $# -eq 0 ]; then
    /usr/games/fortune | /usr/games/cowsay
  else
    /usr/games/cowsay "$@"
fi
```

还需要用 `chmod +x entrypoint.sh` 把文件设为可执行。

注 8：编写 `ENTRYPOINT` 脚本的时候要非常小心，不要把用户给弄糊涂了——请记住任何给 `docker run` 的命令都会传给脚本，但却有可能与用户预期的行为不一致。

如果执行脚本的时候没有提供任何参数，那么脚本只会把 fortune 的输出通过管道传递给 cowsay；否则，cowsay 会依据提供的参数执行。下一步需要更改 Dockerfile，使它把这个脚本放进镜像中，并且通过 ENTRYPOINT 指令调用它。现在修改 Dockerfile 如下：

```
FROM debian

RUN apt-get update && apt-get install -y cowsay fortune
COPY entrypoint.sh /  ❶

ENTRYPOINT ["/entrypoint.sh"]
```

❶ COPY 指令所做的仅仅是把一个文件从主机复制到镜像的文件系统，第一个参数是主机的文件，第二个参数是目标路径，这与 cp 命令类似。

现在再次生成一个新镜像，并尝试以提供参数和不提供参数两种方式运行容器：

```
$ docker build -t test/cowsay-dockerfile .
...snip...
$ docker run test/cowsay-dockerfile
 ____________________________________
/ The last thing one knows in        \
| constructing a work is what to put |
| first.                             |
|                                    |
\ -- Blaise Pascal                   /
 ------------------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
$ docker run test/cowsay-dockerfile Hello Moo
 ___________
< Hello Moo >
 -----------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

3.4 使用寄存服务

刚才我们做了一个很有意思的镜像，如果我们希望与别人分享，该怎么办呢？在本章最初的时候，我们运行的 Debian 镜像是从官方的 Docker 镜像寄存服务（registry）[9]，即 Docker Hub 下载的。同样，我们可以上传自己的镜像到 Docker Hub 供他人下载及使用。

注 9：Docker registry 一词目前还没有一个标准翻译，一般作者或译者选择不翻译。翻译成“寄存服务”的想法来自计算机的寄存器（register）。——译者注

Docker Hub 可以通过命令行或网页访问，你可以使用 Docker 的 search 命令，或者在 http://registry.hub.docker.com 上搜索已有的镜像。

> **寄存服务、仓库、镜像和标签**
>
> 镜像的储存以层级设计，并采用以下术语。
>
> **寄存服务**（registry）
>
> 负责托管和发布镜像的服务，默认为 Docker Hub。
>
> **仓库**（repository）
>
> 一组相关镜像（通常是一个应用或服务的不同版本）的集合。
>
> **标签**（tag）
>
> 仓库中镜像的识别号，由英文和数字组成（如 `14.04` 或 `stable`）。
>
> 举个例子，`docker pull amouat/revealjs:latest` 是指从 Docker Hub 的 `amouat/revealjs` 仓库下载标签为 `latest` 的镜像。

为了能上传我们的 `cowsay` 镜像，必须在 Docker Hub 上注册一个账户（通过网站或者 `docker login` 命令）。注册完成后，只需为镜像指定一个合适名称的仓库和标签，然后用 `docker push` 命令上传到 Docker Hub。不过在上传之前，还要在 Dockerfile 内加入 `MAINTAINER` 指令，这样做是为了给镜像设定作者的联系信息：

```
FROM debian

MAINTAINER John Smith <john@smith.com>
RUN apt-get update && apt-get install -y cowsay fortune
COPY entrypoint.sh /

ENTRYPOINT ["/entrypoint.sh"]
```

现在重新生成镜像，然后上传到 Docker Hub。这一次，仓库名称必须用你的 Docker Hub 账户名开头（我的是 `amouat`），跟着是 `/`，最后以镜像名称结束，镜像名称可以随你的喜好决定。例如：

```
$ docker build -t amouat/cowsay .
...
$ docker push amouat/cowsay
The push refers to a repository [docker.io/amouat/cowsay] (len: 1)
e8728c722290: Image successfully pushed
5427ac510fe6: Image successfully pushed
4a63ead8b301: Image successfully pushed
73805e6e9ac7: Image successfully pushed
c90d655b99b2: Image successfully pushed
30d39e59ffe2: Image successfully pushed
511136ea3c5a: Image successfully pushed
latest: digest: sha256:bfd17b7c5977520211cecb202ad73c3ca14acde6878d9ffc81d95...
```

由于我没有在仓库名称的后面指定标签，它将会自动被赋予 latest 标签。如果你需要指定使用特定标签名称的话，只需在仓库名称后面加上冒号，然后加上要指定的标签即可（例如 docker build -t amouat/cowsay:stable.）。

上传完毕后，任何人都可以用 docker pull 命令下载你的镜像（例如 docker pull amouat/cowsay）。

私有仓库

当然，你可能不希望全世界的人都能够访问你的镜像。如果是这样的话，你有两个选择。你可以向提供私有仓库托管服务的公司付费（可以是 Docker Hub 或者类似的服务，譬如 quay.io），也可以搭建一个你自己的寄存服务。有关私有仓库和寄存服务的信息，参见第 7 章。

镜像的命名空间

服务器上的 Docker 镜像属于三种命名空间的其中之一，可以由镜像的名称判断它属于哪一种。

- 如果镜像名称以字符串和 / 开头，如 amouat/revealjs，那么它属于“用户”命名空间（“user” namespace）。Docker Hub 上的这些镜像都上传自某个用户。例如 amouat/revealjs 是用户 amouat 上传的 revealjs 镜像。任何人都可以自行上传公开的镜像到 Docker Hub，上面已经有了数千个镜像，从好玩但无聊的 supertest2014/nyan 到实用的 gliderlabs/logspout，不一而足。
- 诸如 debian 或 ubuntu 的名称，不包含前缀或 /，属于“根”命名空间（“root” namespace）。它由 Docker 公司所控制，为一些常用的软件及发行版预留在 Docker Hub 上发布的官方镜像。虽然这个命名空间由 Docker 管理，但实际上镜像通常由第三方维护，一般是该软件的供应商（例如 nginx 镜像是由 nginx 公司维护）。大部分常用的软件都提供官方镜像，当你需要镜像的时候，应该先从官方镜像着手。
- 以主机名或 IP 开头的名称，代表该镜像来自第三方的寄存服务（并非 Docker Hub），包括公司或组织自己搭建的寄存服务，或 Docker Hub 的竞争对手，例如 quay.io。举个例子，localhost:5000/wordpress 是指本地的寄存服务中的 WordPress 镜像。

上述的命名空间规则确保用户能清楚辨别镜像的来源；如果你使用的镜像名叫 debian，你就知道它是 Docker Hub 上的官方镜像，而非来自其他寄存服务的 debian 版本。

3.5 使用Redis官方镜像

我得承认，你能从 cowsay 镜像中学到的东西不会太多。那就试试怎样使用官方 Docker 仓库中的镜像吧。接下来看一下 Redis 官方镜像，它是一个很流行的键值存储（key-value store）。

官方仓库

如果你在 Docker Hub 上搜索一个很流行的应用程序或服务，例如 Java 编程语言或 PostgreSQL 数据库，你会发现返回的结果有好几百个。[10] 官方 Docker 仓库的目的是提供确保质量和来源的镜像，其中的镜像都由官方负责把关，所以尽可能将其作为你的首选。搜索的时候它们应该会显示在返回结果的最前面，并标识为官方镜像。

当你从官方仓库下载时，镜像名称不会有用户名的部分，或者会设为 `library`（例如 MongoDB 仓库同时由 `mongo` 和 `library/mongo` 提供）。你还会看到消息说“你正获取的镜像已被核实”（The image you are pulling has been verified），表示 Docker 守护进程已经验证过镜像的校验和，因此已核实它的来源。

首先获取镜像：

```
$ docker pull redis
Using default tag: latest
latest: Pulling from library/redis

d990a769a35e: Pull complete
8656a511ce9c: Pull complete
f7022ac152fb: Pull complete
8e84d9ce7554: Pull complete
c9e5dd2a9302: Pull complete
27b967cdd519: Pull complete
3024bf5093a1: Pull complete
e6a9eb403efb: Pull complete
c3532a4c89bc: Pull complete
35fc08946add: Pull complete
d586de7d17cd: Pull complete
1f677d77a8fa: Pull complete
ed09b32b8ab1: Pull complete
54647d88bc19: Pull complete
2f2578ff984f: Pull complete
ba249489d0b6: Already exists
19de96c112fc: Already exists
library/redis:latest: The image you are pulling has been verified.
Important: image verification is a tech preview feature and should not be re...
Digest: sha256:3c3e4a25690f9f82a2a1ec6d4f577dc2c81563c1ccd52efdf4903ccdd26cada3
Status: Downloaded newer image for redis:latest
```

启动 Redis 容器，但这次需要用 `-d` 参数：

```
$ docker run --name myredis -d redis
585b3d36e7cec8d06f768f6eb199a29feb8b2e5622884452633772169695b94a
```

`-d` 参数让容器在后台运行。Docker 照常启动容器，不一样的是，这次不会把容器的输出打印出来，而只会返回容器 ID，然后就会退出。容器仍然在后台运行，你也可以通过 `docker logs` 命令查看容器的输出。

注 10：编写本书的时候，有 1350 个 PostgreSQL 镜像。

好了，但我们怎样来用它？显然需要通过某种方法与数据库进行连接。但我们没有应用程序来担当客户端的角色，因此干脆利用 redis-cli 工具来完成任务。我们可以在主机上安装 redis-cli，但如果启动一个新的容器来运行 redis-cli，并把这两个容器连接上，这样做不但更简单，而且还更有意思：

```
$ docker run --rm -it --link myredis:redis redis /bin/bash
root@ca38735c5747:/data# redis-cli -h redis -p 6379
redis:6379> ping
PONG
redis:6379> set "abc" 123
OK
redis:6379> get "abc"
"123"
redis:6379> exit
root@ca38735c5747:/data# exit
exit
```

刚才我们把两个容器连在了一起，并只花了几秒钟就已经在 Redis 中添加了数据，既漂亮又简洁。但这是怎么做到的？

Docker 中网络连接的改变

这一章及本书其他部分均使用 --link 命令来把容器连接起来。但在不久的将来，Docker 将会改变网络连接的做法，连接容器会以更易于表述的“发布服务”（publish services）方式取而代之。不过，连接方式在可预见的时期内仍会支持，本书的范例在不修改的情况下应能照常工作。

如果想要进一步了解这个即将发生的变化，参见 11.4 节。

能将两个容器神奇地连接在一起，是通过 docker run 命令的 --link myredis:redis 参数实现的。这个参数告诉 Docker 把新容器与现存的“myredis”容器连接起来，并且在新容器中以“redis”作为“myredis”容器的主机名。为了实现这一点，Docker 会在新容器中的 /etc/hosts 里添加一个新条目，把“redis”指向“myredis”的 IP 地址。这样就能够在执行 redis-cli 的时候直接使用“redis”作为主机名，而不需想办法找出或传递 Redis 容器的 IP 地址给 redis-cli。

然后，执行 Redis 的 ping 命令，核实我们已连上 Redis 服务器，然后才用 set 和 put 命令添加和获取数据。

目前为止，所有事情看起来都还不错，但有个问题：我们怎样才能做数据的持久保存和备份？为此，我们不会使用标准的容器文件系统，而是需要一个能够让容器与主机，或容器与其他容器之间轻松共享数据的方式。Docker 通过*数据卷*（volume）的概念提供了这种方式。数据卷是直接在主机挂载的文件或目录，不属于常规联合文件系统的一部分。这意味着它们允许与其他容器共享，而任何修改都会直接发生在主机的文件系统里。声明一个目录为数据卷有两种方法，第一种是在 Dockerfile 里使用 VOLUME 指令，第二种是在执行 docker run 的时候使用 -v 参数。下列的 Dockerfile 指令以及 docker run 命令，都会在容器中的 /data 创建一个数据卷：

```
VOLUME /data
```

以及

```
$ docker run -v /data test/webserver
```

默认情况下，目录或文件会挂载在主机的 Docker 安装目录之下（通常是 /var/lib/docker/）。执行 `docker run` 命令的时候可以指定用于挂载的主机目录（例如 `docker run -d -v /host/dir:/container/dir test/webserver`）。出于可移植性和安全方面的考虑，主机目录是无法在 Dockerfile 中指定的（该文件或目录在其他系统中可能不存在，而容器不应在未获得明确授权的情况下挂载敏感文件，譬如 etc/passwd）。

那么，我们怎样用它为 Redis 容器做备份呢？假设 `myredis` 容器还在运行，下面我将展示一种备份方法：

```
$ docker run --rm -it --link myredis:redis redis /bin/bash
root@09a1c4abf81f:/data# redis-cli -h redis -p 6379
redis:6379> set "persistence" "test"
OK
redis:6379> save
OK
redis:6379> exit
root@09a1c4abf81f:/data# exit
exit
$ docker run --rm --volumes-from myredis -v $(pwd)/backup:/backup \
        debian cp /data/dump.rdb /backup/
$ ls backup
dump.rdb
```

注意这里用 `-v` 参数挂载一个主机上已知的目录，并通过 `--volumes-from` 将新容器连接至 Redis 数据库目录。

`myredis` 容器使用完毕后，可以把它停止并删掉：

```
$ docker stop myredis
myredis
$ docker rm -v myredis
myredis
```

所有剩下的容器都可以这样删掉：

```
$ docker rm $(docker ps -aq)
45e404caa093
e4b31d0550cd
7a24491027fc
...
```

3.6 总结

本章关于 Docker 的入门知识至此就告一段落了。这有点走马观花，但现在你至少对创建和运行容器有了信心。下一章会带大家深入了解 Docker 的架构，以及一些基本概念。

第 4 章

Docker基本概念

本章将会更详细地讲解 Docker 的基本概念。首先来看一下 Docker 总体的系统架构，包括它所利用的相关技术。之后会对几个概念，包括 Docker 镜像的构建、容器之间的相互连接，以及如何处理数据卷中的内容作更深入的探讨。最后，本章会给出常用 Docker 命令的概览。

> 这一章包含很多参考材料，因此你可以选择只看重点，然后直接进入第 5 章，等需要的时候再回来查看。

4.1　Docker系统架构

为了理解怎样才能够尽量发挥 Docker 的能力，以及了解 Docker 的一些不寻常的行为，对 Docker 系统底层的组成有一个大致的了解将会非常有用。

图 4-1 展示了一个 Docker 系统的主要组成部分。

- 图的中央是 Docker 守护进程，它负责容器的创建、运行和监控，还负责镜像的构建和储存，容器和镜像都在图的右边。Docker 守护进程通过 `docker daemon` 命令启动，一般会交由主机的操作系统负责执行。
- Docker 客户端在图的左边，它通过 HTTP 与 Docker 守护进程通信。默认使用 Unix 域套接字（Unix domain socket）实现，但为了支持远程客户端也可以使用 TCP socket。如果该套接字由 systemd 管理的话，也可以使用文件描述符。由于所有通信都必须通过 HTTP，因此与远程的 Docker 守护进程连接并非难事，为一个编程语言开发一套接口库

也会很简单 [但由于不同功能的实现方式，也会存在个别问题，例如 Dockerfile 需要构建环境的上下文（build context），这个问题在 4.2.1 节中会详细解释]。与守护进程通信的 API 都有清晰的定义和详细文档，使得开发者可以利用这套 API 来开发与守护进程直接通信的程序，而无需通过 Docker 客户端。值得一提的是，Docker 客户端和守护进程是由同一个二进制文件发布的。

- Docker 寄存服务负责储存和发布镜像。默认的寄存服务为 Docker Hub，它托管了数以千计的公共镜像，以及由其负责把关的“官方”镜像。许多组织会搭建自己的寄存服务器，用于储存商业用途和机密的镜像，这样做还可以节省从互联网下载镜像所浪费的时间。如果你有意运营自己的寄存服务，请参考 7.4.1 节的内容 。当 Docker 守护进程收到 `docker pull` 请求之后，便会从寄存服务器下载镜像。而当遇到 `docker run` 请求或 Dockerfile 中的 `FROM` 指令时，假如本地没有它们要求的镜像存在，Docker 守护进程也会自动从服务器下载镜像。

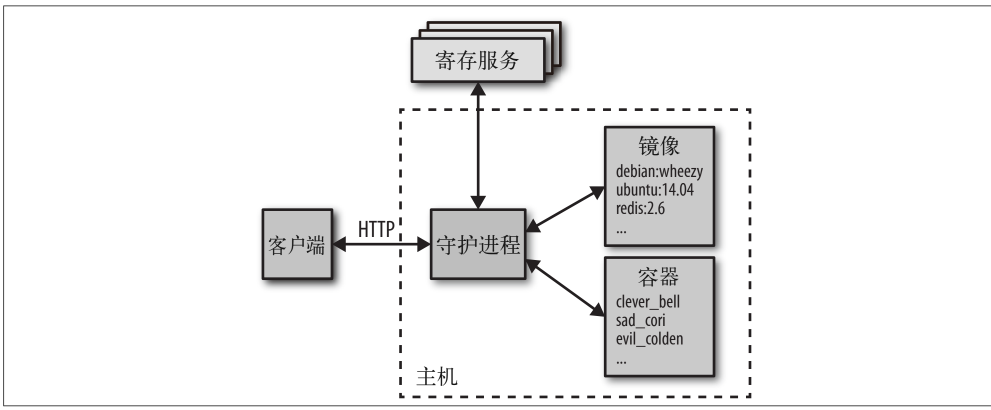


图 4-1：Docker 主要组成部分的总体概览

4.1.1 底层技术

Docker 守护进程通过一个“执行驱动程序”（execution driver）来创建容器。默认情况下，它是 Docker 项目自行开发的 runc 驱动程序，但仍支持旧有的 LXC。runc 与下面提到的内核功能密不可分。

- cgroups，负责管理容器使用的资源（例如 CPU 和内存的使用）。它还负责冻结和解冻容器这两个 `docker pause` 命令所需的功能。
- Namespaces（命名空间），负责容器之间的隔离；它确保系统的其他部分与容器的文件系统、主机名、用户、网络和进程都是分开的。

Libcontainer 还支持 SElinux 和 AppArmor，它们可以给容器更稳固的安全保障。第 13 章会有更详细的介绍。

另一个主要的 Docker 底层技术就是联合文件系统（Union File System，UFS），它负责储存容器的镜像层。UFS 由数个存储驱动中的其中之一提供，可以是 AUFS、devicemapper、BTRFS 或 Overlay。请参考之前 3.3 节“镜像、容器和联合文件系统”中有关 UFS 的论述。

4.1.2 周边技术

Docker 引擎和 Docker Hub 并不足以构成一套完整的容器方案。大部分用户发现它们还需要一些支撑服务和软件，诸如集群管理、服务发现（service discovery）工具和更先进的联网功能。正如在 1.4 节中所描述的，Docker 公司计划打造一个完整的“开箱即用”方案，它除了包括上述功能，还允许用户将默认组件轻松地替换成第三方开发的组件。“可替换电池”（swappable battery）这一策略主要是指 API 层，它使 Docker 引擎的功能可以被其他组件直接调用，但也可以理解为一些已打包成独立二进制档的 Docker 辅助技术，可以很轻松地替换为第三方组件。

以下是 Docker 目前提供的辅助技术。

Swarm

Docker 的集群方案。Swarm 可以把多个 Docker 主机组合起来，使其资源能整合为一体。第 12 章会有更详细的介绍。

Compose

Docker Compose 是负责构建和运行由多个 Docker 容器所组成的应用程序的工具。它主要用于开发和测试，而不太用于生产环境。在 5.2 节中会有更深入的介绍。

Machine

Docker Machine 可以在本地或远程资源上安装和配置 Docker 主机。Machine 还能配置 Docker 客户端，使用户能轻松地在不同的环境之间切换。相关的例子见第 9 章。

Kitematic

Kitematic 是一个 Mac OS 和 Windows 上的 GUI，用于运行和管理 Docker 容器。

Docker Trusted Registry

Docker 的一个企业内部方案，用于储存和管理 Docker 镜像。实际上它是个 Docker Hub 的本地版本，能够与企业或组织现有的安全基础架构集成，并能帮助他们遵守有关数据存储和安全方面的法规。其特点包括指标（metrics）、基于角色的权限控制（Role-Based Access Control，RBAC）和日志，它们全部通过一个管理控制台管理 。这是 Docker 公司目前唯一的非开源产品。

现在已有大量基于或配合 Docker 使用的第三方服务和应用。下面列出在不同领域中出现的一些解决方案。

网络连接

要把不同主机上的容器连接起来并不简单，解决方法也是五花八门。现在，这个领域中已经出现了一些解决方案，包括 Weave（http://weave.works/net/）和 Project Calico（http://www.projectcalico.org/）。而且在不久的将来，Docker 也将实现一个集成的联网方案，名为 Overlay。 通过 Docker 的联网插件架构，用户可以用其他方案取代 Overlay 驱动程序。

服务发现

当 Docker 容器启动时，它需要通过某种方法来找出与之通信的服务，而这些服务一般也是运行在容器内的。容器的 IP 地址是动态分配的，因此在大型系统中要解决这个问题并非易事。 这方面的解决方案包括 Consul（https://consul.io/）、Registrator（https://github.com/gliderlabs/registrator）、SkyDNS（https://github.com/skynetservices/skydns/），以及 etcd（https://github.com/coreos/etcd）。

服务编排及集群管理

在大型的容器部署上，监控和管理系统的工具必不可少。 对每个新的容器，都需要安排把它放置在哪台主机上，并对它实施监控和保持更新。系统需要对故障的出现或负载的改变作出反应，实际情况下可能是搬迁、启动或停止容器。 这方面已有数个解决方案互相竞争，包括来自谷歌的 Kubernetes（http://kubernetes.io/）、Marathon（https://github.com/mesosphere/marathon）[来 自 Mesos（https://mesos.apache.org/） 的 框 架]、CoreOS 的 Fleet（https://github.com/coreos/fleet），以及 Docker 自家的 Swarm。

这些话题都会在第三部分作深入的介绍。值得一提的是，Docker Trusted Registry 也有替代方案， 包括 CoreOS 的 Enterprise Registry（https://coreos.com/products/enterprise-registry/）和来自 JFrog 的 Artifactory（http://www.jfrog.com/open-source/#os-arti）。

除了前面提到的联网驱动插件，Docker 也支持*数据卷插件*（volume plugin），用于与其他存储系统集成。比较著名的数据卷插件有 Flocker（https://github.com/ClusterHQ/flocker），一个多主机数据管理及迁移工具，以及用于分布式存储的 GlusterFS（https://github.com/calavera/docker-volume-glusterfs）。更多有关插件框架的信息可以在 Docker 网站（https://docs.docker.com/engine/extend/legacy_plugins/）找到。

容器的兴起附带引发了一个很有趣的现象，那就是诞生了一些专门用于托管容器的操作系统。虽然 Docker 在大部分现今的 Linux 发行版上都运行得很好，例如在 Ubuntu 和红帽上，但还是出现了一些新的发行版项目，它们只考虑需要运行容器（或容器及虚拟机）的环境，希望做出体积小而容易管理的发行版，尤其是针对数据中心或集群的使用场景。这方面的例子有 Project Atomic（http://www.projectatomic.io/）、CoreOS（https://coreos.com/）和 RancherOS（http://rancher.com/rancher-os/）。

4.1.3 Docker托管

第 9 章将会对 Docker 托管服务作更详细的介绍，现在我们先初步了解有哪些托管服务可供选择。很多传统的云服务供应商，包括亚马逊、谷歌和 Digital Ocean，都已经提供了某种程度的 Docker 服务。谷歌的 Container Engine 可能是其中最有趣的一个，因为它是直接在 Kubernetes 上构造。当然，即使一个云服务供应商没有提供专门的 Docker 支持，一般也可以部署虚拟机，并在其上运行 Docker 容器。

Joyent 也在这方面提供了自己的容器方案，它在 SmartOS 之上构建，叫作 Triton。它利用自己的容器和 Linux 模拟技术实现了 Docker API，使得 Joyent 成功创造了一个可以与普通 Docker 客户端通信的公共云平台。重要的是，Joyent 认为它的容器实现足够安全，可以直

接运行于裸机之上，因此没有必要使用虚拟机，这意味着在效能方面（尤其是输入 / 输出）能得到大大提升。

另外，还有一些 PaaS 平台项目是在 Docker 的基础之上建立的，例如 Deis（http://deis.io/）、Flynn（https://flynn.io/）和 Paz（http://paz.sh）。

4.2 镜像是如何生成的

在 3.3 节中已经介绍过，创建新镜像的主要方法是通过 Dockerfile 和 `docker build` 命令。这一节将会更深入了解它们背后的事情，最后还会介绍 Dockerfile 各指令的用法。学会 `build` 命令的工作原理将会非常有用，因为它的行为有时候确实出乎意料。

4.2.1 构建环境的上下文

`docker build` 命令需要 Dockerfile 和构建环境的上下文（build context，可能是空的）。上下文是一组本地文件和目录，它可以被 Dockerfile 的 `ADD` 或 `COPY` 指令所引用，通常以目录路径的形式指定。举个例子，我们在 3.3 节中曾经使用过命令 `docker build -t test/cowsay-dockerfile .`，它的上下文便是“.”，即当前目录。该目录下的所有文件和目录就形成了构建环境的上下文，并在生成镜像的过程中传递给 Docker 守护进程。

假如没有指定上下文（如果只是提供了 Dockerfile 的 URL，或者 Dockerfile 的内容是通过管道从 `STDIN` 读进来），上下文将被视为空的。

不要使用“/”作为构建环境的上下文

由于构建环境上下文会被放进一个 tar 文件，然后传给 Docker 守护进程，因此你绝对不会希望使用一个含有大量文件的目录。因为客户端需要把所有文件归档，然后传给守护进程，所以像 /home/user、downloads 或 / 之类的目录都需要花费非常多的时间来处理。

如果提供的 URL 以 http 或 https 开头，它会被假定为直接指向 Dockerfile 的链接。这样做其实没什么用，因为该 Dockerfile 没有与任何上下文关联（而且尚未支持指向归档的链接）。

git 仓库也可以作为构建环境的上下文。在这种情况下，Docker 客户端会把仓库和任何子模组（submodule）复制（clone）至一个临时目录，并把它传给 Docker 守护进程作为构建环境的上下文。如果路径以 github.com/、git@ 或 git:// 开头，Docker 便会将路径视为 git 仓库。一般而言，我不建议使用这种方法，比较好的做法是手动把仓库复制到本地，这样做更灵活，而且不容易引起混淆。

Docker 客户端还可以从 `STDIN` 输入构建环境的上下文，方法是在提供上下文的参数处使用“-”。该输入可以是不包含上下文的 Dockerfile（例如 docker build - < Dockerfile），或者是一个包含上下文及 Dockerfile 的归档文件（例如 docker build - < context.tar.gz）。归档文件可以是 tar.gz、xz 或 bzip2 格式。

上下文中 Dockerfile 的位置可以用 `-f` 参数指定（例如 `docker build -f dockerfiles/Dockerfile.debug .`）。若未指定，Docker 则会尝试在上下文的根目录寻找名为 Dockerfile 的文件。

使用 `.dockerignore` 文件

为了从构建环境的上下文中排除不必要的文件，你可以使用 .dockerignore 文件。该文件包含要排除的文件名，文件名以换行符分隔。可以使用 `*` 和 `?` 通配符。举个例子，下面是我们的 `.dockerignore` 文件：

```
.git      ❶
*/.git    ❷
*/*/.git  ❸
*.sw?     ❹
```

❶ 上下文的根目录下的 .git 文件或目录会被排除，但允许它出现在子目录中（即 .git 会被排除，但 dir1/.git 不会）。

❷ 只排除第一层目录下的 .git 文件或目录（即 dir1/.git 会被排除，但 .git 和 dir1/dir2/.git 不会）。

❸ 只排除第二层目录下的 .git 文件或目录（即 dir1/dir2/.git 会被排除，但 .git 和 dir1/.git 不会）。

❹ test.swp、test.swo 和 bla.swp 会被排除，但 dir1/test.swp 不会。

此处不支持完整的正则表达式，例如 `[A-Z]*`。

这本书写作的时候，暂时还没有办法同时匹配所有子目录下的文件（例如，你不能只用一个正则表达式来同时忽略 /test.tmp 和 /dir1/test.tmp）。

4.2.2 镜像层

很多时候，刚接触 Docker 的用户都会被镜像生成的过程难倒。Dockerfile 中的每个指令执行后都会产生一个新的*镜像层*，而这个镜像层其实可以用来启动容器。一个新的镜像层的建立，是用上一层的镜像启动容器，然后执行 Dockerfile 的指令后，把它保存为一个新的镜像。当 Dockerfile 指令成功执行后，中间使用过的那个容器会被删掉，除非提供了 `--rm=false` 参数。[1] 由于每个指令的最终结果都只是个静态的镜像而已，本质上除了一个文件系统和一些元数据以外就没有其他东西，因此即使指令中有任何正在运行的进程，最后都会被停掉。这意味着，虽然你可以在 `RUN` 指令中执行一些持久的进程，譬如数据库或 SSH 服务，但到了处理下一个指令或启动容器的时候，它们就已经不再运行了。如果你需要在启动容器的时候同时运行一个服务或进程，它必须从 `ENTRYPOINT` 或 `CMD` 指令中启动。

你可以通过 `docker history` 命令来查看组成镜像的所有层。例如：

```
$ docker history mongo:latest
IMAGE          CREATED      CREATED BY ...
278372cb22b2   4 days ago   /bin/sh -c #(nop) CMD ["mongod"]
```

注 1：如果我已经把你弄糊涂了，不用担心。看完调试示例中有关 `docker build` 的输出信息就应该能理解了。

```
341d04fd3d27    4 days ago   /bin/sh -c #(nop) EXPOSE 27017/tcp
ebd34b5e9c37    4 days ago   /bin/sh -c #(nop) ENTRYPOINT &{["/entrypoint.
f3b2b8cf226c    4 days ago   /bin/sh -c #(nop) COPY file:ef2883b33ed7ba0cc
ba53e9f50f18    4 days ago   /bin/sh -c #(nop) VOLUME [/data/db]
c537910de5cc    4 days ago   /bin/sh -c mkdir -p /data/db && chown -R mong
f48ad436057a    4 days ago   /bin/sh -c set -x
df59596772ab    4 days ago   /bin/sh -c echo "deb http://repo.mongodb.org/
96de83c82d4b    4 days ago   /bin/sh -c #(nop) ENV MONGO_VERSION=3.0.6
0dab801053d9    4 days ago   /bin/sh -c #(nop) ENV MONGO_MAJOR=3.0
5e7b428dddf7    4 days ago   /bin/sh -c apt-key adv --keyserver ha.pool.sk
e81ad85ddfce    4 days ago   /bin/sh -c curl -o /usr/local/bin/gosu -SL "h
7328803ca452    4 days ago   /bin/sh -c gpg --keyserver ha.pool.sks-keyser
ec5be38a3c65    4 days ago   /bin/sh -c apt-get update
430e6598f55b    4 days ago   /bin/sh -c groupadd -r mongodb && useradd -r
19de96c112fc    6 days ago   /bin/sh -c #(nop) CMD ["/bin/bash"]
ba249489d0b6    6 days ago   /bin/sh -c #(nop) ADD file:b908886c97e2b96665
```

当构建失败时，你可以把失败前的那个层启动起来，这将非常有助于调试。举个例子，我们有下面这个 Dockerfile：

```
FROM busybox:latest

RUN echo "This should work"
RUN /bin/bash -c echo "This won't"
```

现在尝试构建镜像：

```
$ docker build -t echotest .
Sending build context to Docker daemon 2.048 kB
Step 0 : FROM busybox:latest
 ---> 4986bf8c1536
Step 1 : RUN echo "This should work"
 ---> Running in f63045cc086b  ❶
This should work
 ---> 85b49a851fcc  ❷
Removing intermediate container f63045cc086b  ❸
Step 2 : RUN /bin/bash -c echo "This won't"
 ---> Running in e4b31d0550cd
/bin/sh: /bin/bash: not found
The command '/bin/sh -c /bin/bash -c echo"This won't"' returned a non-zero
code: 127
```

❶ Docker 为了执行我们的指令而创建了一个临时容器，这是该容器的 ID。
❷ 这是由该容器所建立的镜像的 ID。
❸ 临时容器在这里被删除。

在这个例子中，虽然从错误信息已经可以明显看出问题所在，但是仍然可以利用最后成功生成的镜像层创建一个镜像出来并执行它，用作调试指令之用。请注意，我们使用的 ID 是最后的镜像的 ID（85b49a851fcc），而不是最后容器的 ID（e4b31d0550cd）：

```
$ docker run -it 85b49a851fcc
/# /bin/bash -c "echo hmm"
/bin/sh: /bin/bash: not found
/# /bin/sh -c "echo ahh!"
```

```
ahh!
/ #
```

现在就可以更清楚地看到问题的原因：busybox 镜像没有包含 bash shell。

4.2.3 缓存

Docker 为了加快镜像构建的速度，也会将每一个镜像层缓存下来。Docker 的缓存特性能大大提高工作效率，可是它的实现却有点笨拙。你的指令必须满足以下条件，Docker 才会使用缓存：

- 上一个指令能够在缓存中找得到，并且
- 缓存中存在一个镜像层，而它的指令与你的指令一模一样，父层也完全相同（即使指令中出现一些无关重要的空格也会使缓存失效）

此外，关于 COPY 和 ADD 指令，如果它们引用的文件的校验和或元数据发生了变化，那么缓存也将失效。

这意味着，即使那些每次调用的结果可能都不一样的 RUN 指令，也仍然会被缓存。如果需要下载文件、执行 apt-get update 或复制源码库，请务必注意这一点。

如果你需要使缓存失效，可以在执行 docker build 的时候加上 --no-cache 参数。你还可以通过添加或修改一个指令，使之后的缓存失效；因此，你可能在一些 Dockerfile 里会看到类似这样的写法：

```
ENV UPDATED_ON "14:12 17 February 2015"
RUN git clone....
```

我不推荐这个方法，因为它会让将来使用这个镜像的用户感到困惑，尤其是镜像创建的日期与其中所写的不一样时。

4.2.4 基础镜像

当你创建自己的镜像时，你需要决定从哪个基础镜像（base image）开始。因为可供选择的镜像非常多，所以花一点时间去了解它们各自的优缺点是很值得的。

最理想的情况是完全不用创建镜像，只需直接使用某个现有的镜像，然后把配置文件和 / 或数据挂载上去即可。对于常见的应用软件，诸如数据库和 Web 服务器，这是非常可行的，因为它们都已经有可用的官方镜像。一般情况下，使用官方镜像要比自己创建一个镜像要好得多，因为其他人已经找到了使得该软件以最佳方式运行在容器中的方法，你能从他们的工作成果和经验中受益。如果有任何特殊原因导致你无法使用官方镜像，可以考虑把问题报告给管理该镜像的项目，因为很有可能其他人也面临着类似的问题，或者已经有回避它的已知方法。

如果你需要一个基础镜像用来运行应用程序，那么应该先查一下应用程序所使用的编程语言或框架（例如 Go 或 Ruby on Rails）是否已提供了官方镜像。通常构建和发布软件可以使用不同的镜像（譬如可以使用 java:jdk 镜像来构建 Java 应用，但在发布 JAR 文件的时候则使用体积较小的 java:jre 镜像，因为它去除了不必要的编译工具）。同样，一些官方

镜像（如 node）还特别提供了已去除大部分开发工具和头文件的“精简”版本。

有时候，你真正需要的可能只是一个小而完整的 Linux 发行版。如果追求极简主义，我会使用 alpine 镜像，它的大小仅仅 5MB 多一点，但仍提供了一个包管理器，可以轻松安装大量应用和工具。如果我需要一个更完整的镜像，我通常会从 debian 系列中选一个，虽然 ubuntu 镜像也很常见，但 debian 镜像比它小，还可以使用相同的软件包。即使你的组织只允许使用某个特定的 Linux 发行版，你应该也能找到它的 Docker 镜像。这样做要比迁移到一个你的组织根本不支持或没有经验的新发行版更合理。

很多时候，过分追求镜像的最小化是没有必要的。请记住，不同镜像会共享相同的基础镜像层，因此，如果你已经有 ubuntu:14.04 镜像，然后从 Hub 下载一个基于它的另一个镜像，那么，你只会下载它在基础镜像之上改变过的部分，而并非整个镜像。不过，细小的镜像确实有利于快速部署和分发。

做出一个极小的镜像还是有可能的，你可以在镜像中只放入二进制文件。为此，Dockerfile 需要继承一个特殊的 scratch 镜像（它是一个完全空白的文件系统），然后只需将你的二进制文件复制进去，并设置适当的 CMD 指令。你的二进制文件必须包含所有需要的程序库（无动态链接），并且不会调用任何外部命令。此外，还要记住编译二进制文件时的目标架构，它必须与容器的架构相同，因为运行 Docker 客户端的机器架构可能与容器的架构不同。[2]

虽然极简方法非常诱人，但不得不注意，当需要进行调试或维护的时候，你将会面临诸多不便，因为 busybox 并没有太多供你使用的工具，要是使用了 scratch 镜像，你甚至连 shell 都没有。

Phusion 镜像的争议

关于基础镜像，一个值得关注的选择是 phusion/baseimage-docker。Phusion 的开发者创建这个基础镜像的原因是出于对 Ubuntu 官方镜像的一些想法，他们声称它缺少了某些关键服务。一些 Docker 的核心开发者并不同意 Phusion 的观点，这导致后来在博客、IRC 以及推特上曾经出现过多番议论。争论的要点如下。

是否需要 init 服务

Docker 的观点是每个容器只该运行一个单一的应用，甚至最好只运行**一个进程**。如果你只有一个进程，那就没有必要使用 init 服务。Phusion 提出的主要论点是，

注 2：这种极简的设计概念其实还可以更进一步，舍弃 Docker 以及整个 Linux 内核，以 unikernel 的做法取而代之。在 unikernel 架构中，应用程序与仅含有该应用程序所需功能的内核相结合，然后由虚拟器管理程序直接执行。这样做可以省去一些不必要的代码和不会使用的驱动，使得应用程序更小也更快（unikernel 通常可以在一秒内启动，也就是说，它可以直接由用户请求启动）。了解更多内容请参阅 Anil Madhavapeddy 和 David J. Scott 的文章“Unikernels: Rise of the Virtual Library Operating System”（https://queue.acm.org/detail.cfm?id=2566628）以及 MirageOS 系统（http://www.openmirage.org/）。

没有 init 服务导致容器内充满僵尸进程——僵尸进程是那些没有被父进程正常结束，或者没有被负责监管的进程收拾干净的进程。虽然这种说法是正确的，但僵尸进程发生的唯一可能是应用程序的代码中存在 bug；绝大多数的用户应该不会遇到这个问题，如果有，最好的解决方法是修复代码。

是否保持 cron 守护进程运行

Ubuntu 和 debian 的基础镜像默认不启动 cron 守护进程，但 phusion 不一样。Phusion 认为许多应用程序都依赖 cron，因此它必须保持运行。而 Docker 的观点，同时也是我倾向认同的观点，就是 cron 在你的应用程序依赖它的时候才需要运行。

SSH 服务

默认镜像都不会安装 SSH 服务或运行它。正常获取 shell 的方法是执行 docker exec 命令（参见 4.6.2 节），否则每个容器都需要运行一个不必要的进程，而这只会带来坏处。Phusion 似乎接受了这一点，并已默认禁用了 SSH 服务，但它的镜像仍然包含 SSH 程序和它的库，使得镜像还是颇为臃肿。

就个人而言，只有需要在容器中运行多个进程、cron 以及 SSH 时，我才会推荐使用 Phusion 的基础镜像。否则，我会坚持使用来自 Docker 官方仓库的镜像，如 ubuntu:14.04 及 debian:wheezy。

重建镜像

请注意，当执行 docker build 的时候，Docker 会查看 FROM 指令所指定的镜像，如果本地没有该镜像，Docker 会试图下载它。如果本地已有该镜像，Docker 就会使用它，而不会先检查是否有更新的可用版本。这意味着仅仅执行 docker build 并不足以保证你的镜像是最新的，你还必须对所有被依赖的父镜像显式地执行 docker pull 或删掉它们，使 build 命令不得不下载最新版本。

鉴于一般的基础镜像（如 debian）都会发布安全补丁更新，因此这一点尤为重要。

4.2.5 Dockerfile指令

这一节对 Dockerfile 的各个指令作扼要介绍。这里不会详述细节，一方面是因为 Docker 还在不断变化，所以内容很可能很快就会过时，另一方面是因为在 Docker 网站（http://docs.docker.com/reference/builder/）内就能够找到全面且最新的文档。Dockerfile 的注释方法是以 # 作为一行的开头。

Exec 与 Shell 格式的对比

一些指令（RUN、CMD 以及 ENTRYPOINT）能够接受 shell 和 exec 这两种格式。exec 格式需要用到一个 JSON 数组（例如，["executable", "param1", "param2"]），其中第一个元素是一个可执行文件，其他元素是它执行时所使用的参数。Shell 格式使用的是自由形式的字符串，字符串会传给 /bin/sh -c 执行。exec 格式适用于需要规避 shell 对字符串作出错误解析的情况，或者当镜像里没有包含 /bin/sh 时。

以下列出的是 Dockerfile 的可用指令。

ADD

从构建环境的上下文或远程 URL 复制文件至镜像。如果是从一个本地路径添加一个归档文件，那么它会被自动解压。由于 ADD 指令涵盖的功能相当广泛，一般最好还是使用相对简单的 COPY 指令来复制构建环境上下文的文件和目录，并用 RUN 指令配合 curl 或 wget 来下载远程资源（这样还可以在同一个指令中处理和删除下载文件）。

CMD

当容器启动时执行指定的指令。如果还定义了 ENTRYPOINT，该指令将被解释为 ENTRYPOINT 的参数（在这种情况下，请确保使用的是 exec 格式）。CMD 指令也会被 docker run 命令中镜像名称后面的所有参数覆盖。假如定义了多个 CMD 指令，那么只有最后一个生效，前面出现过的 CMD 指令全部无效（包括出现在基础镜像中的那些）。

COPY

用于从构建环境的上下文复制文件至镜像。它有两种形式，COPY src dest 以及 COPY ["*src*", "*dest*"]，两者皆从上下文中的 src 复制文件或目录至容器内的 dest。如果路径中有空格的话，那么必须使用 JSON 数组的格式。通配符可以用来指定多个文件或目录。请注意，你不能指定上下文以外的 src 路径（例如 ../another_dir/myfile 是不管用的）。

ENTRYPOINT

设置一个于容器启动时运行的可执行文件（以及默认参数）。任何 CMD 指令或 docker run 命令中镜像名称之后的参数，将作为参数传给这个可执行文件。ENTRYPOINT 指令通常用于提供“启动”脚本，目的是在解析参数之前，对变量和服务进行初始化。

ENV

设置镜像内的环境变量。这些变量可以被随后的指令引用。例如：

```
...
ENV MY_VERSION 1.3
RUN apt-get install -y mypackage=$MY_VERSION
...
```

在镜像中这些变量仍然可用。

EXPOSE

向 Docker 表示该容器将会有一个进程监听所指定的端口。提供这个信息的目的是用于连接容器（参见 4.4 节）或在执行 `docker run` 命令时通过 `-P` 参数把端口发布开来；`EXPOSE` 指令本身并不会对网络有实质性的改变。

FROM

设置 Dockerfile 使用的基础镜像；随后的指令皆执行于这个镜像之上。基础镜像以“镜像：标签”（`IMAGE:TAG`）的格式表示（例如 `debian:wheezy`）。如果省略标签，那么就被视为最新（`latest`），但我强烈建议你一定要给标签设置为某个特定版本，以免出现任何意想不到的事情。`FROM` 必须为 Dockerfile 的第一条指令。

MAINTAINER

把镜像中的“作者”元数据设定为指定的字符串。可以通过 `docker inspect -f {{.Author}} IMAGE` 这个命令来查看该信息。这个指令通常用于设置镜像维护者的姓名和联系方式。

ONBUILD

指定当镜像被用作另一个镜像的基础镜像时将会执行的指令。对于处理一些将要添加到子镜像的数据，这个指令将会非常有用（例如，把代码从一个已选定的目录中复制出来，并在执行构建脚本时使用它）。

RUN

在容器内执行指定的指令，并把结果保存下来。

USER

设置任何后续的 `RUN`、`CMD` 或 `ENTRYPOINT` 指令执行时所用的用户（用户名或 UID）。请注意，UID 在主机和容器中是相同的，但用户名则可能被分配到不同的 UID，导致设置权限时变得复杂。

VOLUME

指定为数据卷的文件或目录。如果该文件或目录已经在镜像中存在，那么当容器启动时，它就会被复制至这个卷。如果提供了多个参数，那么就会被解释成多个数据卷。出于对可移植性和安全性的考虑，你不能在 Dockerfile 中指定数据卷将会使用的主机目录。更多相关信息参见 4.5 节。

WORKDIR

对任何后续的 `RUN`、`CMD`、`ENTRYPOINT`、`ADD` 或 `COPY` 指令设置工作目录。这个指令可多次使用。支持使用相对路径，按上次定义的 `WORKDIR` 解析。

4.3 使容器与世界相连

假设你正在一个容器中运行 Web 服务器。你如何使外界能访问它呢？答案是通过 `-p` 或 `-P`

选项来“发布”端口。这个命令能够将主机的端口转发到容器上。例如：

```
$ docker run -d -p 8000:80 nginx
af9038e18360002ef3f3658f16094dadd4928c4b3e88e347c9a746b131db5444
$ curl localhost:8000
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

其中的 `-p 8000:80` 参数告诉 Docker 将主机的 8000 端口转发至容器的 80 端口。或者，可以使用 `-P` 选项来告诉 Docker 自动选择一个主机上未使用的端口。例如：

```
$ ID=$(docker run -d -P nginx)
$ docker port $ID 80
0.0.0.0:32771
$ curl localhost:32771
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
...
```

`-P` 选项的主要优势是你不再需要负责跟踪哪些端口被分配过，如果你有几个容器同时发布了不同端口，这就变得很重要了。这时候只需要使用 `docker port` 命令来找出被 Docker 分配的端口。

4.4 容器互联

Docker 的连接（link）是允许同一主机上的容器互相通信的最简单方法。当使用 Docker 默认的联网模型时，容器之间的通信将通过 Docker 的内部网络，这意味着主机网络无法看见这些通信。

> **Docker 关于联网的改变**
>
> 在将来的 Docker 版本（可能是 1.9 或以后）中，容器互联的惯用方式将改为“发布服务”的形式，而不再是“连接容器”。然而，连接方式在可预见的时间段内仍会支持，本书的范例在不修改的情况下应仍能照常工作。
>
> 如果要了解更多即将发生的有关联网的改变，参见 11.4 节。

连接的初始化是通过执行 `docker run` 命令时传入 `--link CONTAINER:ALIAS` 参数，其中 `CONTAINER` 是目标容器的名称[3]，而 `ALIAS`（别名）是主容器用来称呼目标容器的一个本地名称。

使用 Docker 的连接也会把目标容器的别名和 ID 添加到主容器中的 /etc/hosts，允许主容器

注 3：在这部分以至全书，我把被连接的容器称为*目标容器*，而容器被启动的一方称为*主容器*（因为它是负责建立连接的一方）。

通过名称找到目标容器。

此外，Docker 还会在主容器内设置一系列的环境变量，目的是为了更方便与目标容器通信。例如，假设我们创建一个 Redis 容器并连接到它：

```
$ docker run -d --name myredis redis
c9148dee046a6fefac48806cd8ec0ce85492b71f25e97aae9a1a75027b1c8423
$ docker run --link myredis:redis debian env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=f015d58d53b5
REDIS_PORT=tcp://172.17.0.22:6379
REDIS_PORT_6379_TCP=tcp://172.17.0.22:6379
REDIS_PORT_6379_TCP_ADDR=172.17.0.22
REDIS_PORT_6379_TCP_PORT=6379
REDIS_PORT_6379_TCP_PROTO=tcp
REDIS_NAME=/distracted_rosalind/redis
REDIS_ENV_REDIS_VERSION=3.0.3
REDIS_ENV_REDIS_DOWNLOAD_URL=http://download.redis.io/releases/redis-3.0.3.tar.gz
REDIS_ENV_REDIS_DOWNLOAD_SHA1=0e2d7707327986ae652df717059354b358b83358
HOME=/root
```

可以看到，Docker 设置了一些以 `REDIS_PORT` 为前缀的环境变量，它们包含如何连接到容器的信息。大部分变量的信息看起来似乎有些多余，因为它们的值跟名称都有重复的内容。尽管如此，即使只把它们看作一种文档记录也很有用。

Docker 还导入了目标容器的环境变量，并在它们的名字前面加上 `REDIS_ENV`。虽然这个做法可以很有用，但如果你利用环境变量来存储机密信息，例如 API 令牌或数据库密码，那么就要倍加注意了。

默认情况下，无论容器之间是否已经建立了显式连接，都可以互相通信。如果想要防止尚未连接的容器能够互联，可以在启动 Docker 守护进程时加上 `--icc=false` 和 `--iptables` 这两个参数。当容器之间建立了连接时，Docker 便会设置 iptables 规则让容器可以在已声明为开放（exposed）的端口上进行通信。

遗憾的是，Docker 的连接功能目前还有一些不足之处。其中最大的不足也许就是连接只能是静态的。虽然连接在容器重启之后应该还能工作，但如果目标容器被替换了，连接是不会更新的。此外，目标容器必须在主容器之前启动，这意味着双向连接是不可能的。

更多有关容器互联的信息，请参阅第 11 章。

4.5　利用数据卷和数据容器管理数据

本节先简单复习一下 Docker 的数据卷，它是一个目录[4]，但并不属于容器 UFS 的一部分（详见 3.3 节的“镜像、容器和联合文件系统”），它只是在主机上被*绑定挂载*（bind mount）到容器的一个普通目录（参见本节的说明“绑定挂载”）。

总共有三种[5]不同的方法进行数据卷的初始化，认识清楚它们之间的差异对你非常重要。

注 4：从技术上讲，可以是目录或文件，因为数据卷也可以是一个文件。

注 5：OK，也可以说是两个半，这要看你怎么数。

首先，可以在执行 Docker 时，通过 -v 选项来宣告一个数据卷：

```
$ docker run -it --name container-test -h CONTAINER -v /data debian /bin/bash
root@CONTAINER:/# ls /data
root@CONTAINER:/#
```

这样，容器中的 /data 目录便成为了一个数据卷。镜像的 /data 目录中的所有文件将被复制到数据卷内。我们可以在主机上打开一个新的 shell，通过执行 docker inspect 命令，找出数据卷在主机上的实际位置：

```
$ docker inspect -f {{.Mounts}} container-test
[{5cad... /mnt/sda1/var/lib/docker/volumes/5cad.../_data /data local  true}]
```

在这个例子中，容器的 /data/ 卷仅仅是一个指向主机中 /var/lib/docker/volumes/5cad…/_data 目录的连接。为了证明这一点，我们可以在主机上添加一个文件到那个目录：[6]

```
$ sudo touch /var/lib/docker/volumes/5cad.../_data/test-file
```

你应该可以立刻在容器中看到它：

```
$ root@CONTAINER:/# ls /data
test-file
```

设置数据卷的第二种方法是通过在 Dockerfile 中使用 VOLUME 指令：

```
FROM debian:wheezy
VOLUME /data
```

这个做法与执行 docker run 时提供 -v /data 参数的效果是相同的。

在 Dockerfile 中设置数据卷权限

很多时候你都需要设置数据卷的权限和它的所有者，或者需要把一些默认数据或配置文件用作卷的初始数据。进行这些操作的时候，有一点必须注意，那就是 Dockerfile 中 VOLUME 指令**之后**的所有指令**不可以**对该数据卷有任何修改。例如，下面的 Dockerfile 不会有预期中的效果：

```
FROM debian:wheezy
RUN useradd foo
VOLUME /data
RUN touch /data/x
RUN chown -R foo:foo /data
```

本来我们希望 touch 和 chown 命令在镜像的文件系统上执行，但实际上，它们是在一个用来创建容器层的临时容器内的数据卷上执行的（详细说明请查看 4.2 节）。当命令结束后，这个卷也会被删除，使这些指令变得毫无意义。

注 6：如果你连接的是远程 Docker 服务，你需要先通过 SSH 登录到远程主机，然后在远程主机上执行这个命令。如果你正使用 Docker Machine（假如你是通过 Docker 工具箱安装 Docker 的话），你也可以执行 docker-machine ssh default 来登入系统。

下面的 Dockerfile 则能够达到我们期望的效果：

```
FROM debian:wheezy
RUN useradd foo
RUN mkdir /data && touch /data/x
RUN chown -R foo:foo /data
VOLUME /data
```

当使用这个镜像来启动一个容器的时候，Docker 会把那个数据卷目录内的所有文件从镜像复制到容器。但如果数据卷是和主机目录绑定的话，那么镜像的数据就不会复制到容器了（这是为了防止主机文件被意外覆写）。

如果由于某些原因，你不能在 RUN 指令中设置卷的权限和它的所有者，那么就需要通过使用 CMD 或 ENTRYPOINT 指令，在容器创建后，以执行脚本的方式来解决。

第三种方法 [7] 将 docker run 命令的 -v 选项用法进行扩展，使到能够具体指明数据卷要绑定的主机目录，命令格式为 -v HOST_DIR:CONTAINER_DIR（其中 HOST_DIR 为主机目录，CONTAINER_DIR 为容器目录），而 Dockerfile 并不支持这样做（因为这是不可移植的，而且还会有安全风险）。这个命令的用法如下：

```
$ docker run -v /home/adrian/data:/data debian ls /data
```

这个例子把主机的 /home/adrian/data 目录挂载到容器的 /data 目录。容器能够使用 /home/adrian/data 目录中已有的任何文件。如果容器已经有 /data 目录，它的内容将会被数据卷所隐藏。与先前的几个命令不同，这次并没有文件会从镜像复制到数据卷，而卷也不会被 Docker 删除（换句话说，假如数据卷是挂载到一个用户选定的目录的话，那么 docker rm -v 是不会把它删除的）。

绑定挂载

使用数据卷时特别指明主机目录（即使用了 -v HOST_DIR:CONTAINER_DIR 这个语法），这个做法一般被称为绑定挂载（Bind Mounting）。这个说法多少会令人误解，因为从技术上而言，所有数据卷都是绑定挂载的，不同的是该挂载点的位置是具体指定的，而不是藏于某个 Docker 管理的目录下。

4.5.1 共享数据

如果需要在主机和一个或多个容器之间共享文件，-v HOST_DIR:CONTAINER_DIR 语法是非常有用的。假设容器使用的是一般通用的镜像，我们可以把配置文件保存在主机上，然后把它们挂载到容器来进行配置。

另一种容器间共享数据的方法是，在运行 docker run 命令时，传入 --volumes-from CONTAINER 参数。例如，我们可以创建一个新的容器，让它能够访问之前的示例中该容器的数据卷，像这样：

注 7：排名的话，它应该与第二种方法同样重要。

```
$ docker run -it -h NEWCONTAINER --volumes-from container-test debian /bin/bash
root@NEWCONTAINER:/# ls /data
test-file
root@NEWCONTAINER:/#
```

需要注意的是，与卷关联的那个容器（在刚才的例子中，就是 container-test）无论正在运行与否，刚才的命令都能使用。只要至少存在着一个容器与卷关联，那么卷就不会被删除。

4.5.2　数据容器

一个常用的做法是创建*数据容器*，这种容器的唯一目的就是与其他容器分享数据。这种方法的主要优点在于，它提供了一个方便的命名空间，使数据卷可以很容易通过 --volumes-from 命令进行加载。

例如，我们可以用以下命令，为 PostgreSQL 数据库创建一个数据容器：

```
$ docker run --name dbdata postgres echo "Data-only container for postgres"
```

这个命令将从 postgres 镜像创建一个容器，并且初始化镜像中定义的所有数据卷，最后执行 echo 命令并退出。[8] 我们没有必要让数据容器一直运行，这样做只会浪费资源。

现在便可以通过 --volumes-from 参数，使其他容器也能够使用这个数据卷。例如：

```
$ docker run -d --volumes-from dbdata --name db1 postgres
```

数据容器的镜像

通常，创建数据容器没有必要使用像 busybox 或 scratch 这种极小的镜像。你只需使用与数据使用方一样的镜像即可。例如，为 PostgreSQL 数据库创建数据容器的话，只需使用 postgres 镜像来做。

使用相同的镜像不会占用任何额外空间，因为你肯定已经下载或创建了用作数据使用方的镜像。这样做也让镜像有机会为容器建立任何初始数据，并确保权限设置正确。

删除数据卷

数据卷只会在满足以下条件的时候被删除：

- 容器被 docker rm -v 命令删除，*或*
- docker run 命令执行时带有 --rm 选项

以及

- 目前没有容器与该数据卷关联
- 该数据卷没有被指定使用主机目录（即没有使用 -v HOST_DIR:CONTAINER_DIR 语法）

注 8：这里可以使用任何能立即退出的命令，但运行 docker ps -a 的时候，echo 命令的输出信息有助于提醒我们容器的目的。另一种方法是使用根本不会启动容器的 docker create 命令来取代 docker run。

目前，除非每次运行容器的时候都谨记以上几点，否则就很有可能在 Docker 的安装目录下积累一些被遗忘的文件和目录，而且很难弄清它们的来历。Docker 正在开发一个名为 volume 的基本命令，它将允许你在不用考虑容器的情况下，对数据卷进行列出、创建、检查以及删除的操作。预计这一功能将随 1.9 版本推出，本书出版的时候应该已经发布。

4.6 Docker常用命令

本节仅针对日常使用的 Docker 命令作扼要介绍（至少和官方文档相比较而言），而不会详尽讲解所有命令。由于 Docker 正迅速发展和不断变化，如果需要某个命令最新最全的信息，请查阅 Docker 网站上的官方文档（http://docs.docker.com）。这里不会罗列和详细介绍各命令的选项和语法（`docker run` 除外）。因此，你可以借助内建的帮助信息，查看方法是在命令之后加上 `--help` 选项，或者通过使用 `docker help` 命令。

Docker 布尔型选项

在大多数 Unix 命令行工具中，你会发现很多选项不需要指定任何值，例如 `ls -l` 中的 `-l`。由于这些选项是要么使用，要么不使用，Docker 视这种选项为布尔型选项，但与大多数程序不一样的是，它还支持对选项明确赋予布尔值（即它能同时接受 `-f=true` 和 `-f` 的用法）。另外，选项的默认值可能是 true 或 false（这也使得事情开始变得复杂）。不同于默认值为 false 的选项，对于默认值为 true 的选项，不赋值等同于给了 true 参数。使用了某选项但没有提供参数，等同于把选项设置成 true——给一个默认为 true 的选项赋值不会改变它的值；要改变它的值，唯一的方法是将其明确设置成 false（例如 `-f=false`）。

要找出一个选项的默认值是 true 还是 false，可以参阅该命令的 `docker help`。例如：

```
$ docker logs --help
...
  -f, --follow=false        跟随日志输出
  --help=false              列出使用方式
  -t, --timestamps=false    显示时间戳
...
```

可以看到，`-f`、`--help` 以及 `-t` 选项都是默认为 false。

下面来看几个关于 `docker run` 命令中默认为 true 的 `--sig-proxy` 选项的实际例子。唯一能把这个选项关掉的方法就是显式地把它设置成 false。例如：

```
$ docker run --sig-proxy=false ...
```

以下所有命令效果相同：

```
$ docker run --sig-proxy=true ...
$ docker run --sig-proxy ...
$ docker run ...
```

若选项默认为 false，如 `--read-only`，以下命令将其设置为 true：

```
$ docker run --read-only=true
$ docker run --read-only
```

执行命令时不提供该选项，或明确地把它设为 false，效果相同。

有些选项具有“短路”逻辑[9]，会导致一些古怪行为（例如 `docker ps --help=false` 能正常工作，但不会打印帮助信息）。

4.6.1 run命令

前文已经展示了 `docker run` 命令的实际操作；启动新容器时必然会用到它。因此，它是迄今为止最复杂的命令，能支持非常多的参数。它的选项允许用户配置镜像运行的方式、覆盖 Dockerfile 设置、配置联网，以及设置容器的权限和资源。

下列选项控制容器的生命周期，以及它的基本运作模式。

`-a, --attach`

把指定的数据流（如 `STDOUT` 之类）连接至终端。若未指定，则默认连接 `stdout` 和 `stderr`。若数据流未指定，而容器以交互模式（`-i`）启动，则 `stdin` 也会被连接至终端。

此选项与 `-d` 选项不兼容。

`-d, --detach`

使容器在“分离”模式下运行。容器会在后台运行，而命令的返回值是容器的 ID。

`-i, --interactive`

保持 `stdin` 打开（即使它没有被被连接至终端[10]）。一般与 `-t` 同时使用，用作启动交互式会话的容器。例如：

```
$ docker run -it debian /bin/bash
root@bd0f26f928bb:/# ls
...省略...
```

`--restart`

配置 Docker 在什么情况下尝试重新启动已退出的容器。参数为 `no` 意味着永远不会尝试重新启动容器；`always` 指不管退出状态是什么，总会尝试重新启动；`on-failure` 仅当退出状态不为 0 的时候才会尝试重启 ，并且可以追加一个可选参数，指定尝试重启的次数，超过重启次数就会放弃（如果没有指定，那就一直重试）。例如，`docker run --restart on-failure:10 postgres` 将启动 postgres 容器，并当退出值不为 0 的时候，尝试重启最多 10 次。

`--rm`

退出时自动删除容器。不能与 `-d` 选项同时使用。

`-t, --tty`

分配一个伪终端（pseudo-TTY）。通常与 `-i` 同时使用，用来启动交互式容器。

注 9：短路逻辑指命令行的某个选项导致该选项后的参数不会被继续解释或执行。——译者注

注 10：如通过 `-a` 或 `--attach` 参数。——译者注

以下选项允许设置容器名称和变量。

`-e, --env`

设置容器内的环境变量。例如：

```
$ docker run -e var1=val -e var2="val 2" debian env
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
HOSTNAME=b15f833d65d8
var1=val
var2=val 2
HOME=/root
```

另外，`--env-file` 选项可以经文件传入环境变量。

`-h, --hostname NAME`

设置容器的 unix 主机名为 `NAME`。例如：

```
$ docker run -h "myhost" debian hostname
myhost
```

`--name NAME`

把 `NAME` 设置为容器的名称。以后，其他 Docker 命令便可以使用该名称来称呼这个容器。

以下选项允许用户进行数据卷的设置（详见 4.5 节）。

`-v, --volume`

这个选项可以用来设置数据卷（数据卷即一个容器中的文件或目录，实际属于主机的文件系统，而非容器的联合文件系统的一部分），有两种形式的参数可供使用。第一种形式仅指定容器中的目录，Docker 会自行选定一个主机上的目录与之绑定。第二种形式除了指定容器目录，还指定与容器目录绑定的主机目录。

`--volumes-from`

挂载指定容器拥有的数据卷。经常用于数据容器（参见 4.5.2 节）。

有数个选项与网络连接有关。以下是经常用到的几个基本命令。

`--expose`

与 Dockerfile 的 `EXPOSE` 指令功能一样。指定容器将会使用的端口或端口范围，但并不会把端口打开。只有与 `-P` 参数同时使用，以及在连接容器时，才有真正意义。

`--link`

建立一个与指定容器连接的内部网络接口。详情参见 4.4 节。

`-p, --publish`

"发布"容器的端口，使主机能访问它。若没有指定主机端口，则会随机分配一个高端口，可通过 `docker port` 命令查看分配了哪个端口。还可以指定端口是在主机的哪个网络接口开放。

`-P, --publish-all`

“发布”所有已指定为开放（exposed）的容器端口，使主机能访问它们。每个容器端口均对应一个随机挑选的高端口。`docker port` 命令可以用来查看端口之间的映射关系。

如果你需要更高级的联网功能，还有几个进阶的选项可用。但请注意，这些选项中有一些要求你对联网有一定了解，以及明白联网在 Docker 中如何实现。欲了解更多详情，请参阅第 11 章。

`docker run` 命令还具有很多选项用来控制容器的权限及性能 。第 13 章有更详细的介绍。

下面的选项会直接覆盖 Dockerfile 中的设置。

`--entrypoint`

把参数指定为容器的入口（entrypoint），覆盖任何 Dockerfile 中的 `ENTRYPOINT` 指令。

`-u, --user`

设置命令运行时所使用的用户。可以以用户名或 UID 指定。此选项会覆盖 Dockerfile 中的 `USER` 指令。

`-w, --workdir`

将参数的路径设置为容器的工作目录。此选项会覆盖 Dockerfile 中的 WORKDIR 指令。

4.6.2　容器管理

在容器的生命周期中，除了 `docker run` 命令外，以下的 `docker` 命令也能用于管理容器。

`docker attach [OPTIONS] CONTAINER`

`attach` 命令允许用户查看容器内的主进程，或与它进行交互。例如：

```
$ ID=$(docker run -d debian sh -c "while true; do echo 'tick'; sleep 1; done;")
$ docker attach $ID
tick
tick
tick
tick
```

这时候，如果你按下 CTRL-C 中断命令的话，不但进程会结束，还会导致容器退出。

`docker create`

从镜像创建容器，但不启动它。与 `docker run` 大部分参数相同。`docker start` 命令可以用来启动容器。

`docker cp`

在容器和主机之间复制文件和目录。

`docker exec`

在容器中运行一个命令。可用于执行维护工作，或替代 `SSH` 用作登入容器。

例如：

```
$ ID=$(docker run -d debian sh -c "while true; do sleep 1; done;")
$ docker exec $ID echo "Hello"
Hello
$ docker exec -it $ID /bin/bash
root@5c6c32041d68:/# ls
bin   dev  home  lib64  mnt  proc  run   selinux  sys  usr
boot  etc  lib   media  opt  root  sbin  srv    tmp  var
root@5c6c32041d68:/# exit
exit
```

`docker kill`

发送信号给容器中的主进程（PID 1）。默认发送 `SIGKILL` 信号，这将导致容器立即退出。另外，发送的信号可以通过 `-s` 选项指定。该命令会返回容器的 ID。

例如：

```
$ ID=$(docker run -d debian bash -c \
    "trap 'echo caught' SIGTRAP; while true; do sleep 1; done;")
$ docker kill -s SIGTRAP $ID
e33da73c275b56e734a4bbbefc0b41f6ba84967d09ba08314edd860ebd2da86c
$ docker logs $ID
caught
$ docker kill $ID
e33da73c275b56e734a4bbbefc0b41f6ba84967d09ba08314edd860ebd2da86c
```

`docker pause`

暂停容器内的所有进程。进程不会接收到关于它们被暂停的任何信号，因此它们无法执行正常结束或清理的程序。进程可以通过 `docker unpause` 命令重启。`docker pause` 的底层利用 Linux 的 cgroup freezer 功能实现。这个命令与 `docker stop` 不同，`docker stop` 会将所有进程停止，并对进程发送信号，让它们察觉得到。

`docker restart`

重新启动一个或多个容器。大致相当于先对容器执行 `docker stop`，然后执行 `docker start`。`-t` 为一个可选参数，它指定一个等待时间，即容器被 `SIGTERM` 信号杀掉之前，让容器有多少时间关闭。

`docker rm`

删除一个或多个容器。返回值是删除成功的容器名称或 ID。默认情况下，`docker rm` 不会删除任何数据卷。`-f` 参数可以用来删除运行中的容器，而 `-v` 参数会删除由容器创建的数据卷（只要它们不是绑定挂载，或正被其他容器使用）。

例如，要删除所有已停止的容器：

```
$ docker rm $(docker ps -aq)
b7a4e94253b3
e33da73c275b
f47074b60757
```

`docker start`

启动一个或多个已停止的容器。可以用来重新启动已退出的容器，或启动由 `docker create` 创建但从未启动的容器。

`docker stop`

停止（但不删除）一个或多个容器。对容器执行 `docker stop` 后，它的状态将转变为“已退出”。`-t` 为一个可选参数，它指定一个等待时间，即容器被 `SIGTERM` 信号杀掉之前，让容器有多少时间关闭 。

`docker unpause`

重启先前被 `docker pause` 命令暂停的容器。

从容器分离

如果你正连接到一个 Docker 容器，无论容器是以交互模式启动，还是通过使用 `docker attach`，当你试图以 CTRL-C 断开时，容器也会同时停止。但如果使用 CTRL-P CTRL-Q 的话，就可以从容器分离，而不会停止容器。

这个方法只有在附有 TTY 的交互模式下才有效（即同时使用了 `-i` 和 `-t` 选项）。

4.6.3　Docker信息

下面的子命令可以获取更多有关 Docker 安装和使用方法的信息。

`docker info`

打印 Docker 系统和主机的各种信息。

`docker help`

把一个子命令作为参数，打印有关该子命令的使用方法和帮助信息。相当于运行命令时提供 `--help` 参数。

`docker version`

打印 Docker 客户端和服务器版本，以及编译时使用的 Go 版本。

4.6.4　容器信息

以下命令提供更多有关运行中及已停止的容器信息 。

`docker diff`

对比容器所使用的镜像，显示容器的文件系统的变化。例如：

```
$ ID=$(docker run -d debian touch /NEW-FILE)
$ docker diff $ID
A /NEW-FILE
```

`docker events`

打印守护进程的实时事件。键入 Ctrl-C 退出。欲了解更多信息，请参阅第 10 章。

`docker inspect`

把容器或镜像作为参数，获取它们的详细信息。这些信息包括大部分配置信息、联网设置以及数据卷的映射信息 。这个命令接受 `-f` 参数，让用户提供一个 Go 模板，对输出结果进行格式编排和信息过滤。

`docker logs`

输出容器的“日志”，也就是曾经输出到容器中的 `STDERR` 或 `STDOUT` 的内容。更多有关 Docker 日志记录的信息，请参阅第 10 章。

`docker port`

把容器作为参数，列出它的端口映射信息。还可以指定要查看的容器内部端口和协议。常用于执行 `docker run -P <image>` 命令之后查看已分配的端口。

例如：

```
$ ID=$(docker run -P -d redis)
$ docker port $ID
6379/tcp -> 0.0.0.0:32768
$ docker port $ID 6379
0.0.0.0:32768
$ docker port $ID 6379/tcp
0.0.0.0:32768
```

`docker ps`

提供关于当前容器的高阶信息，例如名称、ID 和状态。这个命令支持很多不同参数，其中值得一提的是 `-a` 参数，它可以用来获取所有容器的信息，而不仅仅是运行中的容器。还有 `-q` 参数，它使得这个命令只返回容器的 ID，对于用作其他命令如 `docker rm` 的输入非常有用。

`docker top`

把容器作为参数，提供该容器内运行中进程的信息。实际上，这个命令是在主机上运行 UNIX 的 `ps` 命令，然后把容器以外的进程过滤掉。这个命令接受与 `ps` 命令相同的参数，默认为 `-ef`（但请注意，PID 字段必须出现在输出里）。

例如：

```
$ ID=$(docker run -d redis)
$ docker top $ID
UID   PID   PPID  C  STIME  TTY  TIME      CMD
999   9243  1836  0  15:44  ?    00:00:00  redis-server *:6379
$ ps -f -u 999
UID  PID   PPID  C  STIME  TTY       TIME  CMD
999  9243  1836  0  15:44  ?     00:00:00  redis-server *:6379
$ docker top $ID -axZ
LABEL           PID   TTY  STAT   TIME  COMMAND
docker-default  9243  ?    Ssl    0:00  redis-server *:6379
```

4.6.5 镜像管理

下面的命令用于镜像的创建和处理。

`docker build`

从 Dockerfile 建立镜像。关于使用这个命令的详情，参见 3.3 节和 4.2 节。

`docker commit`

从指定的容器创建镜像。虽然 `docker commit` 在创建镜像时很有用，但由于 `docker build` 的可重复性强，因此一般情况下还是比较推荐使用后者。默认情况下，容器在创建镜像前会先暂停，但这个行为可以用 `--pause=false` 选项禁止。这个命令接受 `-a` 和 `-m` 参数来设定元数据。

例如：

```
$ ID=$(docker run -d redis touch /new-file)
$ docker commit -a "Joe Bloggs" -m "Comment" $ID commit:test
ac479108b0fa9a02a7fb290a22dacd5e20c867ec512d6813ed42e3517711a0cf
$ docker images commit
REPOSITORY  TAG   IMAGE ID      CREATED              VIRTUAL SIZE
commit      test  ac479108b0fa  About a minute ago   111 MB
$ docker run commit:test ls /new-file
/new-file
```

`docker export`

将容器的文件系统中的内容以 tar 归档的格式导出，并输出到 `STDOUT`。生成的归档可以通过 `docker import` 导入。请注意，它只会导出文件系统；任何元数据，如映射端口、`CMD` 和 `ENTRYPOINT` 配置将会丢失。另外，数据卷也不会包含在导出归档中。你可以将这个命令与 `docker save` 作对比。

`docker history`

输出镜像中每个镜像层的信息。

`docker images`

列出所有本地镜像，包括库名称、标签名称以及镜像大小等信息。默认情况下，中间镜像（用于创建最上一层的镜像）不会列出。`VIRTUAL SIZE` 是镜像和它下面的所有镜像层的总大小。由于这些镜像层可以与其他镜像共享，简单地把所有镜像的大小加起来并不能准确估算实际磁盘的使用情况。此外，如果镜像有多个标签，那么镜像会重复列出；你可以通过比较 ID 来分辨不同的镜像。这个命令能够接受几个参数，尤其值得一提的是 `-q`，它使命令只返回镜像 ID，方便用作其他命令如 `docker rmi` 的输入。

例如：

```
$ docker images | head -4
REPOSITORY             TAG     IMAGE ID      CREATED       VIRTUAL SIZE
identidock_identidock  latest  9fc66b46a2e6  26 hours ago  839.8 MB
redis                  latest  868be653dea3  6 days ago    110.8 MB
containersol/pres-base latest  13919d434c95  2 weeks ago   401.8 MB
```

如要删除所有被遗留的镜像（dangling image）[11]，可以使用以下命令：

```
$ docker rmi $(docker images -q -f dangling=true)
Deleted: a9979d5ace9af55a562b8436ba66a1538357bc2e0e43765b406f2cf0388fe062
```

docker import

从一个含有文件系统的归档文件创建镜像，归档可以由 docker export 产生。归档可以通过文件路径或 URL 指定，或者通过 STDIN 流导入（使用“-”参数）。命令的返回值是新创建镜像的 ID 。可以通过提供仓库和标签名称来为镜像附加标签。要注意的是，通过 import 生成的镜像只会有一个镜像层，并会失去所有的 Docker 配置信息，如已指定为开放的端口和 CMD 指令的值。你可以将这个命令与 docker load 相比较。

下面的例子用先导出后导入的方法，将一个镜像原来所有的镜像层合为一个。

```
$ docker export 35d171091d78 | docker import - flatten:test
5a9bc529af25e2cf6411c6d87442e0805c066b96e561fbd1935122f988086009
$ docker history flatten:test
IMAGE         CREATED         CREATED BY  SIZE      COMMENT
981804b0c2b2  59 seconds ago              317.7 MB  Imported from -
```

docker load

加载仓库，仓库以 tar 归档的形式从 STDIN 读入。仓库可以包含数个镜像和标签 。与 docker import 不同，该镜像还包含历史和元数据。适用的归档文件可以通过 docker save 创建，这使得 save 和 load 能成为寄存服务器以外用于分发镜像及备份的方案。相关例子参见 docker save。

docker rmi

删除指定的一个或多个镜像。镜像可以用 ID 或仓库加标签名称的方式来指定。如果指定了仓库名称，但没有提供标签名，那么标签会被默认为 latest。要删除存在于多个仓库的镜像，需要用 ID 指定镜像，并同时使用 -f 参数，而且需要对每个仓库分别执行。

docker save

把指定的镜像或仓库储存到 tar 归档，并输出到 STDOUT（如要写入文件，可以使用 -o 选项）。镜像可以用 ID 或 repository:tag 的方式指定。如果只指定了仓库名称，则该仓库中的所有镜像将会被储存到归档，而不仅仅是带 latest 标签的镜像。与 docker load 结合使用，可以用来分发或备份镜像。

例如：

```
$ docker save -o /tmp/redis.tar redis:latest
$ docker rmi redis:latest
Untagged: redis:latest
Deleted: 868be653dea3ff6082b043c0f34b95bb180cc82ab14a18d9d6b8e27b7929762c
...
```

注 11：dangling image 指的是一些已经没有用处的镜像，它们也没有被其他镜像引用，情况类似编程语言里的 dangling pointer 或 dangling reference。docker 没有垃圾回收机制，因此没用的镜像会一直存在，浪费磁盘空间。——译者注

```
$ docker load -i /tmp/redis.tar
$ docker images redis
REPOSITORY          TAG                 IMAGE ID            CREATED
VIRTUAL SIZE
redis               latest              0f3059144681        3 months ago
111 MB
```

`docker tag`

将镜像与一个仓库和标签名称关联。镜像可以通过 ID 或仓库加标签的方式指定（如未提供标签名，默认为 `latest`）。如果没有为新名称提供标签，也默认为 `latest`。

例如：

```
$ docker tag faa2b75ce09a newname ❶
$ docker tag newname:latest amouat/newname ❷
$ docker tag newname:latest amouat/newname:newtag ❸
$ docker tag newname:latest myregistry.com:5000/newname:newtag ❹
```

❶ 把 ID 为 `faa2b75ce09a` 的镜像添加到仓库 `newname`，因为没有指定标签名，所以标签默认为 `latest`。

❷ 把 `newname:latest` 镜像添加到 `amouat/newname` 仓库，这一次也是使用 `latest` 标签。这个名称的格式适用于把镜像推送到 Docker Hub，假设用户为 `amouat`。

❸ 与上一个命令一样，除了标签不再是 `latest`，而是 `newtag`。

❹ 将 `newname:latest` 镜像添加到 `myregistry.com/newname` 仓库，并使用 `newtag` 作为标签。这个名称的格式适用于把镜像推送到位于 http://myregistry.com:5000 的寄存服务器。

4.6.6 使用寄存服务器

以下的命令与使用包括 Docker Hub 在内的寄存服务器有关。要注意的是，Docker 把用户凭证保存在你的主目录中的 .dockercfg 文件：

`docker login`

在指定的寄存服务器进行注册或登录。如果未指定服务器，则假设为 Docker Hub。如果有需要，程序将会要求你提供一些详细信息，你也可以通过参数提供这些信息。

`docker logout`

从 Docker 寄存服务器注销。如果未指定服务器，则假定为 Docker Hub。

`docker pull`

从寄存服务器下载指定的镜像。寄存服务器由镜像名称决定，默认为 Docker Hub。若没有提供标签名，则下载标签为 `latest` 的镜像（如该标签可用）。通过 `-a` 参数可以下载仓库中所有镜像。

`docker push`

将镜像或仓库推送到寄存服务器。如果没有指定标签，则仓库中的所有镜像都会推送到服务器，而不仅仅是标记为 `latest` 的镜像。

`docker search`

列出 Docker Hub 上匹配搜索词的公共仓库。限制结果为最多 25 个仓库。过滤条件还可以包括最低星级以及镜像是否自动生成。通常，在网站上搜索会是最便利的。

4.7 总结

这一章的信息量实在是太丰富了！ 即使只是大概浏览一下要点，你也能了解不少 Docker 的工作原理和主要的命令。在第二部分你将会看到如何将这些知识应用到软件项目中，从开发阶段一直到生产环境。通过动手实践，你将更容易理解这一章的内容。

第二部分

Docker与软件生命周期

第一部分介绍了容器背后的理念以及基本操作。第二部分将使用 Docker 来构建、测试和部署一个 Web 应用程序，以便更深入地了解 Docker。我们将看到如何在开发、测试和生产环境中应用 Docker 容器。这一章将集中讲述单一主机的系统，而有关在多主机环境中如何部署和编排容器的内容将在第三部分介绍。

第二部分结束后，你将明白如何把 Docker 整合到软件开发过程中，并能在日常中自如地使用 Docker。为了最大限度地利用 Docker 的功能，学会采用 DevOps 的思路是很重要的。特别是在开发过程中，我们会思考在生产环境中软件将如何运行，这样做能够减轻部署到各种环境时将要面临的痛苦。

虽然在后面的章节里我们将要建立的应用程序很小，但是其中涵盖了运行由大型开发团队维护的规模庞大的应用时所需的技术和实践。

容器不适合构建那种发布周期以周或月为单位的大型单一架构企业软件。相反，容器天然适合采用微服务的方式，以及探索诸如持续部署（continuous deployment）这样的技术，使得我们能安全地在一天内多次更新生产环境。

容器、DevOps、微服务以及持续交付（continuous delivery）的优势归根结底源于快速循环反馈的想法。通过更快速的迭代，我们可以在更短的时间内开发、测试和验证更高质量的系统。

第 5 章

在开发中应用Docker

第二部分将开发一个简单的 Web 应用程序，这个程序能够对传入的字符串返回一个唯一的图像，类似于 GitHub 和 Stack Overflow 网站对那些没有设定头像的用户使用的 identicons 技术[1]。我们将会利用 Python 语言和 Flask Web 框架编写这个程序。我选择 Python 作为这个示例的编程语言，因为它很常用，而且简洁易懂。即使你不懂 Python 编程也不用担心，我们只会集中讨论怎样与 Docker 交互，而不是研究 Python 代码的细节。[2] 同样，选择 Flask 的原因是它很轻量且易懂。Docker 将会用来管理所有依赖关系，因此完全不需要在主机上安装 Python 或 Flask。

在进入下一章的开发阶段前，本章先来让读者熟悉基于容器的工作流程，并把需要的工具准备妥当。

5.1　说声“Hello World!”

首先，创建一个只返回“Hello World!”的 Web 服务。第一步，先创建一个名为 identidock 的新目录作为我们的项目目录。在这个目录里，创建一个名为 app 的子目录，Python 代码将存放于此。在 app 目录下，创建一个名为 identidock.py 的文件：

```
$ tree identidock/
identidock/
└── app
    └── identidock.py

1 directory, 1 file
```

注 1：identicon 是一种基于用户信息的散列值生成图像的技术。——译者注

注 2：如果想要了解更多关于 Python 和 Flask 的知识，尤其是打算构造 Web 应用的话，请参阅由 Miguel Grinberg 撰写的《Flask Web 开发：基于 Python 的 Web 应用开发实战》一书。（该书已由人民邮电出版社出版，书号：978-7-115-37399-1。——编者注）

将下面的代码放入 identidock.py：

```
from flask import Flask
app = Flask(__name__) ❶


@app.route('/') ❷
def hello_world():
    return 'Hello World!\n'

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0') ❸
```

现在简单解释一下这段代码。

❶ 对 Flask 进行初始化，以及建立一个应用程序的对象。
❷ 创建一个与指定的 URL 相关的 route。每当这个 URL 收到请求，`hello_world` 函数便会被调用。
❸ 初始化 Python 的 Web 服务器。这里使用 `0.0.0.0`（而非 `localhost` 或 `127.0.0.1`）作为指定主机的参数，因为我们需要把它绑定到所有网络接口，否则这个容器无法被主机或其他容器访问到。前一行的 `if` 语句确保此行只有在当文件是以独立程序运行时才会执行，如果这个文件只是一个大型应用的一部分，那么这一行是不会被执行的。

源代码

本章的源代码可以在 GitHub（https://github.com/using-docker/using_docker_in_dev）上找到。这一章中每个阶段的代码都有对应的 git 标签。

据我所知，从电子书复制 / 粘贴代码可能会有问题，如果你遇到类似情况，请使用 GitHub 上的源码库。

现在需要一个容器，把代码放进去并执行它。在 identidock 目录下，创建一个名为 Dockerfile 的文件，并写入以下内容：

```
FROM python:3.4

RUN pip install Flask==0.10.1
WORKDIR /app
COPY app /app

CMD ["python", "identidock.py"]
```

这个 Dockerfile 使用了已安装好 Python 3 的 Python 官方基础镜像。在镜像之上，它会安装 Flask，并把我们的代码复制进去。`CMD` 命令只运行我们的 identidock 代码。

官方镜像的变种

许多流行的编程语言，例如 Python、Go 和 Ruby，它们的官方仓库中都有多个镜像以供不同用途。除了不同版本的镜像，你可能还会发现以下的变种中的至少一个。

slim

这种镜像是标准镜像的精简版本。它们缺少很多常见的软件包和库。当你为了发布而需要减少镜像的大小时，这样做是理所当然的，但如果刚好需要用到那些从标准镜像中移除了的软件包的话，安装和维护它们往往会带来额外的负担。

onbuild

这种镜像使用 Dockerfile 的 `ONBUILD` 指令，它会把某些命令延后至继承这个 onbuild 镜像的“子镜像”中执行，而执行的时刻便是新建这个子镜像的时候。这些命令具体被处理的时刻是子镜像的 `FROM` 指令执行的时候，而通常做的事情包括把代码复制进来以及运行编译步骤。这些镜像对于一门编程语言来说，可以让用户更快和更容易上手，但长远来看，它们往往有诸多限制，而且容易造成混乱。我一般不会推荐使用 onbuild 镜像，除非是第一次对它的镜像库进行探索。

在我们的应用程序示例中，使用的是一个标准的 Python 3 基础镜像，而非它的任何一个变种。

现在，可以构建和运行我们的简单应用了：

```
$ cd identidock
$ docker build -t identidock .
...
$ docker run -d -p 5000:5000 identidock
0c75444e8f5f16dfe5aceb0aae074cc33dfc06f2d2fb6adb773ac51f20605aa4
```

在这里，我把 `-d` 选项传给 `docker run`，让它在后台启动容器，但如果想看到 Web 服务器的输出，也可以把它省略。`-p 5000:5000` 参数告诉 Docker，我们要将容器的 5000 端口转发到主机上的 5000 端口。

现在来测试一下：

```
$ curl localhost:5000
Hello World!
```

Docker Machine 的 IP 地址

如果你是通过 Docker Machine 来运行 Docker 的话（譬如曾使用 Mac 或 Windows 的 Docker 工具箱来安装 Docker），你将无法使用 `localhost` 作为 URL，而是必须使用 Docker 虚拟机的 IP 地址。通过 Docker machine 的 `ip` 命令可以帮助我们把这个步骤自动化。例如：

```
$ curl $(docker-machine ip default):5000
Hello World!
```

本书假设 Docker 是在本地运行，因此务必在有需要的时候将 `localhost` 换成合适的 IP 地址。

非常好！ 但是，目前这个工作流程有一个比较严重的问题：即使代码只有少许改变，我们也需要重新创建镜像，并且重启容器。幸好，有一个简单的解决方法。我们可以把主机上的源码目录绑定挂载（bind mount）到容器内的源码目录之上。下面的代码将停止并删除上次运行的容器（假如刚才的例子不是最后运行的容器，请通过 `docker ps` 找出它的 ID），然后代码会挂载到 /app，并且启动一个新的容器：

```
$ docker stop $(docker ps -lq)
0c75444e8f5f
$ docker rm $(docker ps -lq)
$ docker run -d -p 5000:5000 -v "$PWD"/app:/app identidock
```

`-v "$PWD"/app:/app` 参数把位于 /app 的 app 目录挂载到容器内。它将覆盖容器中 /app 目录的内容 ，而且在容器内还可以进行读写（如果你不希望这样，也可以把数据卷挂载为只读）。参数 `-v` 必须是绝对路径，因此在这个例子中，我们在当前的目录前加上 `$PWD`，不仅可以节省键入的字数，还能提高可移植性。

绑定挂载

假如你使用 `docker run` 的时候，通过 `-v HOST_DIR:CONTAINER_DIR` 参数来指定数据卷使用的主机目录，这个做法一般被称为“绑定挂载”，因为它将主机上的目录（或文件）与容器中的目录（或文件）绑定。这个称呼或许会造成一些混淆，因为所有数据卷技术上都是绑定挂载的，只不过当没有明确指定主机目录的时候，我们需要花点工夫把目录找出来。

注意，主机目录（`HOST_DIR`）指运行 Docker 引擎的机器。如果你远程连接到 Docker 服务的话，那么远端的计算机上必须已经存在该路径。如果你正在使用由 Docker machine 部署的本地虚拟机（如果你通过 Docker 工具箱来安装 Docker），Docker 会把你的 home 目录互相挂载，使你的开发过程更轻松。

验证一下我们的容器仍在工作：

```
$ curl localhost:5000
Hello World!
```

虽然我们刚才挂载的目录，与镜像中 `COPY` 命令所添加的目录所用的路径相同，但现在这个目录在主机上和容器上都是同一个，而并非从镜像复制得来的。正因为如此，现在我们可以在主机上编辑 identidock.py，并能立即看到变化：

```
$ sed -i '' s/World/Docker/ app/identidock.py
$ curl localhost:5000
Hello Docker!
```

这里我用 sed 工具对 identidock.py 文件做了一个快速的原地更改。如果你的机器上没有安装 sed，或者你对它不熟悉，你可以用任何一个文本编辑器，直接把“World”改成“Docker”。

现在，我们有了一个相当标准的开发环境，除了程序依赖的东西——包括 Python 编译器以及程序库，其他的东西都已封装在 Docker 容器中。然而，仍有一个关键问题有待解决。现在还无法在生产环境中使用这个容器，主要原因是它运行的是默认的 Flask Web 服务器，

而它仅适合于开发环境，在实际执行时，它的效率很低，而且不够安全。采用 Docker 的一个关键因素在于，它能减少开发和生产环境之间的差异，来看看如何能做到这一点。

什么？没有 virtualenv ？

如果你是一个经验丰富的 Python 开发者，你可能会对没有使用 virtualenv（https://virtualenv.pypa.io/en/latest/）来开发我们的程序而感到惊讶。对于隔离多个 Python 环境，virtualenv 是个非常有用的工具 。它允许开发者为每个应用程序配置不同的 Python 版本和所需的程序库。一般而言，它在开发 Python 程序中扮演着非常重要的角色，使用率也相当高。

但当你使用容器后，它就变得没有那么有用了，因为我们已经有了一个隔离环境。如果已经习惯了使用 virtualenv，你当然可以继续在容器中使用它，但它能带来的好处或许就不多了，除非你遇到容器内的其他应用程序或库造成的冲突。

uWSGI（https://uwsgi-docs.readthedocs.org/en/latest/）是一个可立即用于生产环境的应用服务器，它还可以部署在 Web 服务器（例如 nginx）的后面。使用 uWSGI 而非默认的 Flask Web 服务器，能使我们的容器非常灵活，适用于各种环境。只需改动 Dockerfile 中的两行代码，便可将容器过渡到 uWSGI：

```
FROM python:3.4

RUN pip install Flask==0.10.1 uWSGI==2.0.8 ➊
WORKDIR /app
COPY app /app

CMD ["uwsgi", "--http", "0.0.0.0:9090", "--wsgi-file", "/app/identidock.py", \
     "--callable", "app", "--stats", "0.0.0.0:9191"] ➋
```

➊ 添加 uWSGI 到 Python 包的安装列表。

➋ 创建一个运行 uWSGI 的新命令。这里我们告诉 uWSGI 启动一个监听 9090 端口的 HTTP 服务器，并从 /app/identidock.py 运行 app 应用。它还在 9191 端口启动一个数据统计服务器。其实我们还可以在运行 `docker run` 命令时，重新定义 `CMD` 的内容。

现在让我们构建并运行它，看看有什么区别：

```
$ docker build -t identidock .
...
Successfully built 3133f91af597
$ docker run -d -p 9090:9090 -p 9191:9191 identidock
00d6fa65092cbd91a97b512334d8d4be624bf730fcb482d6e8aecc83b272f130
$ curl localhost:9090
Hello Docker!
```

如果现在执行 `docker logs` 并赋予容器 ID，那么会看到 uWSGI 的日志信息，这样就可以确认我们的确是在运行 uWSGI 服务器。此外，我们还要求 uWSGI 披露一些统计数据，这些数据可以在 http://localhost:9191 看到。由于程序不是直接在命令行中执行，那段通常用于启动默认 Web 服务器的 Python 代码并没有执行。

虽然现在服务器可以如常工作，但仍有一些清理工作要做。如果检查一下 uWSGI 的日志，会发现服务器明确指出它正以 root 身份运行。这是一个毫无意义的安全漏洞，要解决它其实非常容易，只需在 Dockerfile 中指定用于运行服务器的用户即可。与此同时，我们还会显式地声明容器监听的端口：

```
FROM python:3.4

RUN groupadd -r uwsgi && useradd -r -g uwsgi uwsgi ❶
RUN pip install Flask==0.10.1 uWSGI==2.0.8
WORKDIR /app
COPY app /app

EXPOSE 9090 9191 ❷
USER uwsgi ❸

CMD ["uwsgi", "--http", "0.0.0.0:9090", "--wsgi-file", "/app/identidock.py", \
     "--callable", "app", "--stats", "0.0.0.0:9191"]
```

新加的几行指令解释如下。

❶ 创建 `uwsgi` 用户和用户组，这是 Unix 系统惯常的做法。
❷ 使用 `EXPOSE` 指令声明主机和其他容器可以访问的端口。
❸ 以 uwsgi 用户执行这一行之后的所有指令（包括 CMD 和 `ENTRYPOINT`）。

容器内的用户和用户组

Linux 内核使用 UID 和 GID 来识别用户，并决定他们的访问权限。将 UID 和 GID 映射到标识符是由操作系统的用户空间处理的。正因为如此，容器中的 UID 和主机上的 UID 是相同的，但在容器内创建的用户和用户组并不会传播到主机。这个做法的坏处是会让访问权限变得混乱：相同的文件，在容器内和容器外显示的拥有者有可能不一样。下面的例子给出了相同的文件，但它的拥有者却改变了：

```
$ ls -l test-file
-rw-r--r--    1 docker   staff 0 Dec 28 18:26 test-file
$ docker run -it -v $(pwd)/test-file:/test-file
debian bash
root@e877f924ea27:/# ls -l test-file
-rw-r--r-- 1 1000 staff 0 Dec 28 18:26 test-file
root@e877f924ea27:/# useradd -r test-user
root@e877f924ea27:/# chown test-user test-file
root@e877f924ea27:/# ls -l /test-file
-rw-r--r-- 1 test-user staff 0 Dec 28 18:26 /test-file
root@e877f924ea27:/# exit
exit
docker@boot2docker:~$ ls -l test-file
-rw-r--r--    1 999      staff 0 Dec 28 18:26 test-file
```

与平时一样构建镜像，然后测试新的用户设置：

```
$ docker build -t identidock .
...
```

```
$ docker run identidock whoami
uwsgi
```

请注意，我们已经把原来调用 Web 服务器的 CMD 指令替换为 `whoami` 命令，而这个命令会返回它在容器内运行时的用户名称。

> **永远谨记设置 USER**
>
> 在任何 Dockerfile 中，把 USER 设置妥当都是非常重要的（也可以在 ENTRYPOINT 或 CMD 脚本内改变执行用户）。如果你不这样做，容器中的进程将会以 root 身份运行。由于容器和主机的 UID 是共享的，假如攻击者成功入侵容器，他就能获得主机的 root 权限。
>
> 目前正在进行把容器的 root 用户自动映射到主机上一个高 UID 用户的工作，但撰写本书的时候（Docker 版本 1.8），这个功能尚未完成。[3]

太好了，现在容器内的命令便不会再以 root 身份运行了。让我们再次启动容器，但这次稍稍改变一下使用的参数：

```
$ docker run -d -P --name port-test identidock
```

这一次没有指定主机上绑定哪些端口。相反，我们使用 -P 参数，Docker 会随机选择一些高端口，并自动将它们映射到容器的每一个已声明为“exposed”的端口。在能够访问容器的服务之前，需要询问 Docker，它随机选择的端口是什么：

```
$ docker port port-test
9090/tcp -> 0.0.0.0:32769
9191/tcp -> 0.0.0.0:32768
```

可以看到，它把 9090 端口与主机的 32769 端口绑定，而 9191 端口则与主机的 32768 端口绑定，现在我们知道通过哪些端口来进行访问了（注意，你的端口很可能和这里的不一样）：

```
$ curl localhost:32769
Hello Docker!
```

起初，这样做似乎是毫无意义的额外步骤，而在这个示例中，这样做的确没有必要，但当你需要在一台主机上运行多个容器的时候，让 Docker 自动寻找并映射到未使用的端口，比你自己跟踪哪些端口更轻松。

那么，我们现在已经把一个与生产环境非常接近的 Web 服务运行起来了。对一个实际的生产环境而言，还有很多东西需要调整，诸如 uWSGI 中关于进程和线程的选项，但是对比那个默认用于调试的 Python Web 服务器，我们现在的差距已经缩小了很多。

可是，现在要面对一个新的问题：我们失去了一些开发工具，例如调试输出，以及由默认的 Python Web 服务器提供的实时重新加载代码的功能。虽然我们可以大大缩小开发环境和生产环境之间的差异 ，但它们仍然有着根本不同的需求，这将导致在不同环境下运行时无可避免地需要一些更改。理想情况下，我们还是希望在开发和实际应用时能够使用相同

注 3：Docker 1.10 已实现将 root 用户映射到主机的一个普通用户，但不是默认行为。——译者注

的镜像，按照所在的运行环境，在启用的功能上作出微调。为了实现这一目标，我们可以通过环境变量和一个简单的脚本，按实际的运行环境切换不同功能。

在 Dockerfile 的同一目录下，创建一个名为 cmd.sh 的脚本，并写入以下内容：

```
#!/bin/bash
set -e

if [ "$ENV" = 'DEV' ]; then
  echo "Running Development Server"
  exec python "identidock.py"
else
  echo "Running Production Server"
  exec uwsgi --http 0.0.0.0:9090 --wsgi-file /app/identidock.py \
             --callable app --stats 0.0.0.0:9191
fi
```

这个脚本的含义应该相当清楚了。如果 `ENV` 变量设定为 `DEV`，那么它将运行调试用的 Web 服务器；否则使用生产服务器。[4] 其中的 `exec` 命令 的目的是为了避免创建一个新进程，以确保 uwsgi 进程能够收到所有信号（如 `SIGTERM`），而不是被父进程所拦截。

善用配置文件和辅助脚本

为了简单起见，我将所有东西放在 Dockerfile 内。但是，随着应用程序的增长，还是尽量把它的内容移到辅助文件和脚本比较好。尤其是 pip 的依赖关系应该移到 requirements.txt 文件，而 uWSGI 的配置则可以移到一个 .ini 文件。

接下来需要更新 Dockerfile，使它能用上这个脚本：

```
FROM python:3.4

RUN groupadd -r uwsgi && useradd -r -g uwsgi uwsgi
RUN pip install Flask==0.10.1 uWSGI==2.0.8
WORKDIR /app
COPY app /app
COPY cmd.sh / ❶

EXPOSE 9090 9191
USER uwsgi

CMD ["/cmd.sh"] ❷
```

❶ 把脚本放进容器。
❷ 从 CMD 指令调用它。

在我们尝试这个新版本之前，是时候把那些仍在运行的旧容器停掉。下面的命令将停止并删除主机上的所有容器。如果有任何容器你仍希望保留，那么请千万不要运行这个命令：

注 4：现在有一些变量，例如端口号，在不同的文件中重复出现。我们可以通过使用参数或环境变量来解决这个问题。

```
$ docker stop $(docker ps -q)
c4b3d240f187
9be42abaf902
78af7d12d3bb


$ docker rm $(docker ps -aq)
1198f8486390
c4b3d240f187
9be42abaf902
78af7d12d3bb
```

现在可以重建附有这个脚本的镜像，然后进行测试：

```
$ chmod +x cmd.sh
$ docker build -t identidock .
...
$ docker run -e "ENV=DEV" -p 5000:5000 identidock
unning Development Server
 *Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
 *Restarting with stat
```

很好。现在当我们以 `-e "ENV=DEV"` 运行的话，就得到一个开发用的服务器；否则得到的是一个用于生产环境的服务器。

开发用服务器

你可能会发现，在开发过程中默认的 Python 服务器无法满足你的需求，尤其是当你需要把多个容器连接起来时。若是如此，则可以在开发时使用 uWSGI。不过你仍然需要切换环境的能力，使你可以打开一些生产环境中不应使用的 uWSGI 功能，譬如实时重新加载代码。

5.2 通过Compose实现自动化

最后，还有一些东西可以自动化，使事情变得更简单。Docker Compose（http://docs.docker.com/compose/）旨在迅速建立和运行 Docker 开发环境。大体上，它使用 YAML 文件来存储不同容器的配置，节省开发者重复且容易出错的输入，以及避免了自行开发解决方案的负担 。因为我们的应用很简单，所以它不会为我们带来太多好处，但当你的应用开始变得复杂时，它就能发挥所长了。Compose 将使我们免于自己维护用于服务编排的脚本，包括启动、连接、更新和停止容器。

如果你的 Docker 是使用 Docker 工具箱来安装的话，你应该已经安装了 Compose。如果没有，请按照 Docker 网站（http://docs.docker.com/compose/install/）的指示进行安装。我在本章中所使用的 Compose 版本为 1.4.0，但因为我们用到的只是基本功能，所以 1.2 以后的版本已足够。

请在 `identidock` 目录中创建一个名为 docker-compose.yml 的文件，并写入以下内容：

```
identidock: ❶
  build: .  ❷
ports:    ❸
 - "5000:5000"
environment: ❹
  ENV: DEV
volumes: ❺
  - ./app:/app
```

❶ 第一行声明构建的容器名称。一个 YAML 文件中可以定义多个容器（在 Compose 的术语中称为服务）。

❷ 这里的 `build` 关键字告诉 Compose，这个容器的镜像是通过当前目录（.）下的 Dockerfile 构建。每个容器的定义必须包括一个 `build` 或 `image` 关键字。`image` 关键字的值，是用于启动容器的镜像的标签或 ID，与 `docker run` 命令的参数相同。

❸ `ports` 关键字相当于 `docker run` 命令的 `-p` 参数，用于声明对外开放的端口。这里我们把容器的 5000 端口映射到主机的 5000 端口。列出端口时可以不带引号，但最好避免这样做，因为当遇到像 56:56 这种值的时候，YAML 会把它解析为以 60 为基数的六十进制数字。

❹ `environment` 关键字相当于 `docker run` 命令的 `-e` 参数，用来设置容器的环境变量。为了运行用于开发的 Flask Web 服务器，这里我们把 `ENV` 变量设成 `DEV` 。

❺ `volumes` 关键字相当于 `docker run` 的 `-v` 参数，用于配置数据卷。这里与之前的做法一样，将 app 目录通过绑定挂载的方式挂载到容器，以便让我们能够从主机修改代码。

还有更多的关键字可以在 Compose 的 YAML 文件中设置，通常它们都与 `docker run` 的参数有一对一的关系。

如果现在运行 `docker-compose up`，会得到与之前执行 `docker run` 命令几乎一模一样的结果：

```
$ docker-compose up
Creating identidock_identidock_1...
Attaching to identidock_identidock_1
identidock_1 | Running Development Server
identidock_1 |  * Running on http://0.0.0.0:5000/
identidock_1 |  * Restarting with reloader
```

到另一个终端下执行：

```
$ curl localhost:5000
Hello Docker!
```

当应用程序运行完毕，可以键入 `ctrl-c` 来停止容器。

如要切换到 uWSGI 服务器，我们需要更改 YAML 文件中的 `environment` 和 `ports` 值。可以通过编辑现有的 docker-compose.yml 文件，或者专门为生产环境创建一个新的 YAML 文件，并在执行 `docker-compose` 时使用 `-f` 参数或 `COMPOSE_FILE` 环境变量指定它。

使用Compose的工作流程

下面是使用 Compose 时常用的命令。大多数命令都不言自明，而且 Docker 也有同样名字的命令，不过还是值得在这里提一下。

`up`

启动所有在 Compose 文件中定义的容器，并且把它们的日志信息汇集一起。通常会使用 `-d` 参数使 Compose 在后台运行。

`build`

重新建造由 Dockerfile 构建的镜像。除非镜像不存在，否则 `up` 命令不会执行构建的动作，因此需要更新镜像时便使用这个命令。

`ps`

获取由 Compose 管理的容器的状态信息。

`run`

启动一个容器，并运行一个一次性的命令。被连接的容器会同时启动，除非用了 `--no-deps` 参数。

`logs`

汇集由 Compose 管理的容器的日志，并以彩色输出。

`stop`

停止容器，但不会删除它们。

`rm`

删除已停止的容器。不要忘记使用 `-v` 参数来删除任何由 Docker 管理的数据卷。

一个普通的工作流程以 `docker-compose up -d` 命令启动应用程序开始。`docker-compose logs` 和 `ps` 命令可以用来验证应用程序的状态，还能帮助调试。

修改代码后，先执行 `docker-compose build` 构建新的镜像，然后执行 `docker-compose up -d` 取代运行中的容器。注意，Compose 会保留原来容器中所有旧的数据卷，这意味着即使容器更新后，数据库和缓存也依旧在容器内（这很可能会造成混淆，因此要特别小心）。如果你修改了 Compose 的 YAML 文件，但不需要构建新镜像，可以通过 `up -d` 参数使 Compose 以新的配置替换容器。如果想要强制停止 Compose 并重新创建所有容器，可以使用 `--force-recreate` 选项来达到目的。

当你不再需要使用该应用时，可以执行 `docker-compose stop` 来停止应用程序。假设代码没有变更，可以通过 `docker-compose start` 或 `up` 来重启相同的容器。使用 `docker-compose rm` 彻底把容器删除。

关于所有命令的完整概览，请参阅 Docker 网站上的参考信息页（https://docs.docker.com/compose/reference/）。

5.3 总结

现在我们已经拥有适合工作的环境，可以用来开发我们的应用程序了。本章中我们学习到了很多东西，包括：

- 在主机上无需安装任何工具的情况下，如何利用官方镜像快速创建一个可移植和可重复创建的开发套件
- 如何通过使用数据卷来动态修改容器中的代码
- 如何在容器中同时维护生产环境和开发环境
- 如何通过使用 Compose 将开发流程自动化

Docker 给了我们一个熟悉的开发环境，以及所需要的一切工具；同时，它也提供了一个与生产环境极其相似的环境，我们可以在其中进行快速测试。

我们还有很多事情要做，特别是关于测试和持续集成 / 交付，在接下来的几章里，我们会逐步在开发过程中学会这些技巧。

第 6 章

创建一个简单的Web应用

本章将会把“Hello World!”程序转变为一个简单的 Web 应用，当用户输入文字时，能够生成一幅独一无二的图像。这些图像有时被称为 identicon，它们都是唯一的，由用户名或 IP 地址所产生，可用于识别用户。学完本章，你将得到一个基本可用的应用，并在后面的章节中继续扩展它，对它进行各种试验。通过创建这个应用，我们将会看到如何把 Docker 容器组建成一个功能全面的系统，并如何自然地发展至微服务架构。

identicon

identicon 是基于某个值而自动产生的图像，这个值一般是 IP 地址或用户名的散列值。这个图像提供了某个对象的一个视觉表达，使它易于识别。其用途包括：通过计算用户名或 IP 地址的散列值，在网站上提供用于识别用户的图像，以及自动生成网站的 favicon。

identicon 于 2007 年初由 Don Park 始创，他为自己的博客开发出一套用于识别评论者身份的程序，程序代码仍然可以在该项目的 GitHub 页面（https://github.com/donpark/identicon）找到。

从那以后，identicon 有过几次使用不同的图形样式的改进。Stack Overflow 和 GitHub（见图 6-1，左）是产生 identicon 的两大网站，它们都把 identicon 应用于尚未设置自定义头像的用户。Stack Overflow 使用由 Gravatar 服务生成的图像（见图 6-1，右）。[1]GitHub 则自行生成图像。

注 1：而 Gravatar 背后使用的技术则来自 WP_Identicon（http://scott.sherrillmix.com/blog/blogger/wp_identicon/）等其他项目。

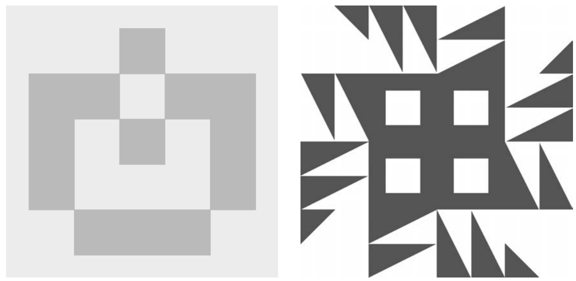


图 6-1：（左）典型的 GitHub identicon；（右）典型的 Gravatar identicon

如果你已经跟着上一章一起做，那么你现在应该有一个类似以下结构的项目：

```
identidock/
├── Dockerfile
├── app
│   └── identidock.py
├── cmd.sh
└── docker-compose.yml
```

如果你没有跟着做也不用担心。你可以在本书的 GitHub 页面（https://github.com/using-docker/creating-a-simple-web-app）获得代码。例如：

```
$ git clone -b v0 https://github.com/using-docker/creating-a-simple-web-app/
...
```

除此之外，你也可以在 GitHub 项目的发布页面上下载代码文件。

`v0` 标签代表代码在上一章结尾时的版本；当我们不断往前推进时，标签也会不断更新。

版本控制

本书假设你具备使用 Git 推送（push）和复制（clone）仓库的知识。后面的章节还会介绍 Docker Hub 与 GitHub 和 BitBucket 如何结合使用。如果你还不是很懂得 Git 的使用，可以查阅 https://try.github.io，那里有免费的教程供学习。

6.1　创建一个基本网页

作为创建应用的第一步，让我们先建立一个很基本的网页。为简单起见，我们将只以字符串的形式返回 HTML。[2] 把 identidock.py 替换成以下的内容：

```
from flask import Flask

app = Flask(__name__)
```

注 2：一个更好的方案是使用模板引擎，譬如 Flask 自带的 Jinja2。

```
default_name = 'Joe Bloggs'


@app.route('/')
def mainpage():

    name = default_name

    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
              Hello <input type="text" name="name" value="{}">
              <input type="submit" value="submit">
              </form>
              <p>You look like a:
              <img src="/monster/monster.png"/>
              '''.format(name)
    footer = '</body></html>'

    return header + body + footer

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

说实话，这和之前的“Hello World!”程序相比差异并不大。我们只是把返回的文本改成一小段 HTML 的页面，其中包括用户输入名字的表格。`format` 函数会把子字符串 `"{}"` 替换为 `name` 变量的值，我们暂且把变量的值设定为“Joe Bloggs”。

执行 `docker-compose up -d`，然后打开浏览器访问地址 http://localhost:5000，可以看到如图 6-2 所示的页面。



图 6-2：首次打开 identidock 页面的样子

图像无法显示是预料之中的，因为还没有加入任何生成图像的代码。同样，提交按钮也无法使用。

开发到了这个阶段，把自动化测试甚至是持续集成 / 交付结合到开发过程当中是个明智之举。不过，为了叙述方便起见，我们会先多做一点开发，然后在接下来的章节中引入测试和持续集成。

6.2　利用现有镜像

现在是时候让我们的程序发挥真正的功用了。我们需要一个函数或服务，输入一个字符串就能输出并返回一个独一无二的图像。有了这个功能之后，我们就可以根据用户在网页上提供的名字，通过调用它来获得一个图像，然后把损坏的图像替换掉。

这个示例中会使用一个现有的 Docker 镜像 dnmonster，它恰恰具备了刚才我所说的功能，

而且还提供了（差不多的）RESTful API 供我们使用。我们还可以用另一个 identicon 服务将 dnmonster 轻松替换掉，特别是如果那个服务也提供了 RESTful API 并已包装成容器。

从已有的代码中调用它，我们需要做一些修改，主要是添加一个新的 `get_identicon` 函数：

```
from flask import Flask, Response ❶
import requests ❷

app = Flask(__name__)
default_name = 'Joe Bloggs'

@app.route('/')
def mainpage():

    name = default_name

    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
              Hello <input type="text" name="name" value="{}">
              <input type="submit" value="submit">
              </form>
              <p>You look like a:
              <img src="/monster/monster.png"/>
              '''.format(name)
    footer = '</body></html>'

    return header + body + footer

@app.route('/monster/<name>')
def get_identicon(name):

    r = requests.get('http://dnmonster:8080/monster/' + name + '?size=80') ❸
    image = r.content

    return Response(image, mimetype='image/png') ❹

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

❶ 导入 Flask 的 `Response` 模块，用它来返回图像。

❷ 导入 Requests 库（http://docs.python-requests.org/en/latest/），用于与 dnmonster 服务通信。

❸ 发送一个 HTTP `GET` 请求到 dnmonster 服务。我们希望得到一个对应 `name` 变量值的 identicon，而它的大小是 80 像素。

❹ 这里的 `return` 语句稍微有点复杂，因为我们需要用 `Response` 函数来告诉 Flask，我们返回的是一个 PNG 图像，而不是 HTML 或文本。

接下来需要对 Dockerfile 做一个小修改，使我们的新代码能使用正确的程序库：

```
FROM python:3.4

RUN groupadd -r uwsgi && useradd -r -g uwsgi uwsgi
RUN pip install Flask==0.10.1 uWSGI==2.0.8 requests==2.5.1 ❶
WORKDIR /app
COPY app /app
```

```
COPY cmd.sh /

EXPOSE 9090 9191
USER uwsgi

CMD ["/cmd.sh"]
```

❶ 添加了前面的 Python 代码中用到的 Requests 库。

现在已经准备就绪，可以启动 dnmonster 容器，然后让它连接到应用程序容器。为了弄清楚背后发生的事情，我们会先用普通的 Docker 命令操作一次，之后才会转用 Compose。由于这是我们第一次使用 dnmonster 镜像，需要从 Docker Hub 下载它：

```
$ docker build -t identidock .
...
$ docker run -d --name dnmonster amouat/dnmonster:1.0
Unable to find image 'amouat/dnmonster:1.0' locally
1.0: Pulling from amouat/dnmonster
...
Status: Downloaded newer image for amouat/dnmonster:1.0
e695026b14f7d0c48f9f4b110c7c06ab747188c33fc80ad407b3ead6902feb2d
```

现在我们会使用与前一章几乎相同的方法启动应用容器，只是多了参数 `--link dnmonster:dnmonster`，这个参数的目的是把两个容器相连。这使得可以在 Python 代码里通过地址 http://dnmonster:8080 访问 dnmonster 服务：

```
$ docker run -d -p 5000:5000 -e "ENV=DEV" --link dnmonster:dnmonster identidock
16ae698a9c705587f6316a6b53dd0268cfc3d263f2ce70eada024ddb56916e36
```

有关连接的详细信息，参见 4.4 节。

如果再次打开浏览器访问 http://localhost:5000，应该会看到类似图 6-3 的画面。



图 6-3：第一个 identicon

虽然它看起来不怎么样，但这是我们的第一个 identicon。提交按钮还未生效，因此我们还没有真正使用任何用户输入，不过很快就会解决这个问题。我们先把 Compose 重拾起来，不然还要记住那些 `docker run` 命令。按照以下内容更新 docker-compose.yml：

```
identidock:
  build: .
  ports:
   - "5000:5000"
  environment:
    ENV: DEV
  volumes:
```

```
    - ./app:/app
  links: ❶
   - dnmonster

dnmonster: ❷
  image: amouat/dnmonster:1.0
```

❶ 声明一个从 identidock 容器到 dnmonster 容器的连接。Compose 会负责处理容器的正确启动顺序，使连接能成功建立。
❷ 定义一个新的 dnmonster 容器。我们只需告诉 Compose，使用来自 Docker Hub 的 `amouat/dnmonster:1.0` 镜像。

启动应用程序之前，我们需要移除之前启动过的任何容器：

```
docker rm $(docker stop $(docker ps -q))
 ...
```

注意，这一指令会停止所有运行中的容器，并不仅限于 identidock 容器。接着使用 Compose 来重建和运行应用程序：

```
$ docker-compose build
...
$ docker-compose up -d
...
```

现在应用程序应该可以重新运行了，并且无需重启容器即可更新代码。

要使按钮生效，我们需要处理发送到服务器的 `POST` 请求，并使用 `form` 变量（它包含了用户名）以生成图像。另外，我们对用户输入的处理过于简单，因此我打算对用户的输入进行散列处理。这样做，可以把任何类似电子邮件地址的敏感输入信息匿名化，还能确保输入是适用于 URL 的格式（即不再需要对诸如空格等字符进行转义）。在我们的应用中，是否使用散列值并不重要，因为处理的只是名字，但这一用法展示了如何在其他情况下利用这个服务，以及如何保护任何偶然输入敏感信息的人。

更新 identicon.py 的内容如下：

```
from flask import Flask, Response, request
import requests
import hashlib ❶

app = Flask(__name__)
salt = "UNIQUE_SALT" ❷
default_name = 'Joe Bloggs'


@app.route('/', methods=['GET', 'POST']) ❸
def mainpage():

    name = default_name
    if request.method == 'POST': ❹
        name = request.form['name']

    salted_name = salt + name
```

```
    name_hash = hashlib.sha256(salted_name.encode()).hexdigest()❺

    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
              Hello <input type="text" name="name" value="{}">
              <input type="submit" value="submit">
              </form>
              <p>You look like a:
              <img src="/monster/{1}"/>
              '''.format(name, name_hash) ❻
    footer = '</body></html>'

    return header + body + footer

@app.route('/monster/<name>')
def get_identicon(name):

    r = requests.get('http://dnmonster:8080/monster/' + name + '?size=80')
    image = r.content

    return Response(image, mimetype='image/png')

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

❶ 导入对用户输入进行散列处理的程序库。由于它是一个标准库，无需修改 Dockerfile 就可以安装它。

❷ 定义散列函数的 `salt` 值。通过改变这个值，相同的输入在不同的网站可以生成不一样的 identicon。

❸ Flask 的 route 默认只会响应 HTTP `GET` 的请求。因为我们的表单提交的是 HTTP `POST` 请求，所以必须给 route 的声明加入以 `methods` 命名的参数，明确宣告 route 可以处理 `POST` 和 `GET` 两种请求。

❹ 如果 `request.method` 等同于 `"POST"`，该请求必定来自点击提交按钮。在此情况下，需要把用户输入的值赋予 `name` 变量。

❺ 对输入执行 SHA256 算法，以获取散列值。

❻ 用散列值改变图像的 URL。这将使得浏览器在加载图像的时候，以散列值调用 `get_identicon`。

保存刚才的修改后，调试用的 Python Web 服务器就能看到程序的改变并自动重启。现在终于可以一睹我们完整的 Web 应用，看看你的 identicon 长什么样子吧（见图 6-4）。

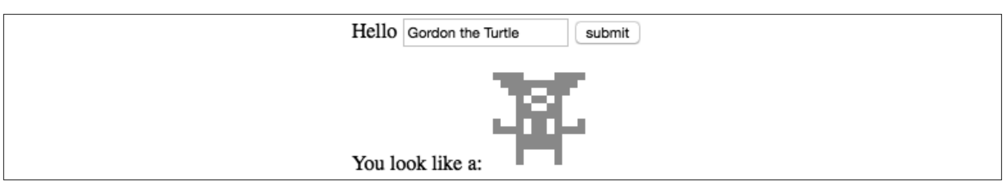


图 6-4：很像小乌龟 Gordon 的 identicon[3]

注 3：Gordon 是 Docker 公司在办公室里饲养的宠物乌龟。——译者注

dnmonster

dnmonster 镜像是一个封装在 Docker 容器中的 Node.js 应用程序。这个应用程序移植自 Kevin Guadin 的 monsterid.js（https://github.com/KevinGaudin/monsterid.js），原本以浏览器的 JavaScript 运行，移植后以 Node.js 运行。而 Monsterid.js 本身是基于 Andreas Gohr 的 MonsterID（http://www.splitbrain.org/projects/monsterid），它能够创建像 RetroAvatar 中的那些 8 位机像素风格的怪兽。你可以在 GitHub（https://github.com/amouat/dnmonster）上找到 dnmonster 项目。

与 monsterid.js 不同，dnmonster 不会对输入进行任何散列处理，而只会把决定权交给调用程序（见图 6-5）。

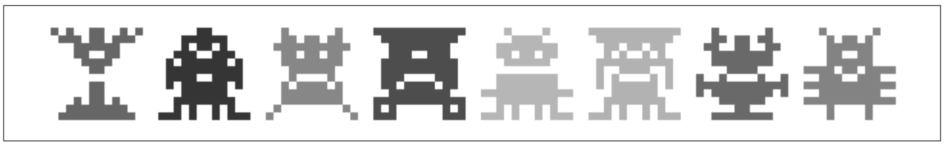


图 6-5：怪兽

6.3 实现缓存功能

目前为止，一切都很顺利。但是，目前我们的应用还有一个令人头疼的事情（除了怪兽的样子），那就是每请求一个怪兽，都需要调用一次 dnmonster 服务，而每次调用都涉及大量运算。这样做其实没有必要。identicon 的原意就是对于相同的输入，图像维持不变，因此我们应该把结果缓存起来。

我们将使用 Redis 帮助我们实现这一目标。Redis 是一个存储于内存的键值数据库。它非常适合像我们这种没有庞大信息量的任务，我们也不用担心数据的持久性（如果某个条目丢失或删掉，只需重新生成图像就好）。可以把 Redis 服务器添加到我们的 identidock 容器，但启动一个新的容器会更容易，而且更符合 Docker 的使用习惯。这样就可以利用 Docker Hub 上已有的官方 Redis 镜像，也就避免了在容器中运行多个进程的麻烦。

在一个容器中运行多个进程

大多数容器只运行一个进程。当需要运行多个进程时，最好还是使用多个容器，并把它们连接起来，如同这个示例中的做法一样。

不过，有时候你真的需要在一个容器内运行多个进程。若是如此，你最好使用进程管理器来负责进程的启动和监视，例如 supervisord（http://supervisord.org/）或 runit（http://smarden.org/runit/）。你也可以写一个简单的脚本来负责进程的启动，但你必须清楚，这样做的话，你便需要处理进程结束后的善后工作，还要负责转发信号。

有关在容器内使用 supervisord 的更多信息，请参阅这篇 Docker 文章：https://docs.docker.com/engine/admin/using_supervisord/。

首先需要更新我们的 Python 代码，使它能够使用缓存：

```
from flask import Flask, Response, request
import requests
import hashlib
import redis ❶

app = Flask(__name__)
cache = redis.StrictRedis(host='redis', port=6379, db=0) ❷
salt = "UNIQUE_SALT"
default_name = 'Joe Bloggs'


@app.route('/', methods=['GET', 'POST'])
def mainpage():

    name = default_name
    if request.method == 'POST':
        name = request.form['name']

    salted_name = salt + name
    name_hash = hashlib.sha256(salted_name.encode()).hexdigest()
    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
              Hello <input type="text" name="name" value="{}">
              <input type="submit" value="submit">
              </form>
              <p>You look like a:
              <img src="/monster/{1}"/>
              '''.format(name, name_hash)
    footer = '</body></html>'

    return header + body + footer


@app.route('/monster/<name>')
def get_identicon(name):

    image = cache.get(name) ❸
    if image is None: ❹
        print ("Cache miss", flush=True) ❺
        r = requests.get('http://dnmonster:8080/monster/' + name + '?size=80')
        image = r.content
        cache.set(name, image) ❻

    return Response(image, mimetype='image/png')

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

❶ 导入 Redis 模块。
❷ 建立 Redis 缓存。我们将通过 Docker 连接的特性，使 `redis` 这个主机名能够被解析。
❸ 检查名字是否已经在缓存中。

❹ 如果缓存未命中，Redis 将返回 None。在这种情况下，程序就像往常一样获取 identicon，不过我们还会……
❺ 输出一些调试信息，表示我们没有在缓存中找到图像，以及……
❻ 把图像添加到缓存中，并与名字相关联。

由于我们引入了一个新模块和新容器，也就不得不同时更新 Dockerfile 和 docker-compose.yml。首先是 Dockerfile：

```
FROM python:3.4

RUN groupadd -r uwsgi && useradd -r -g uwsgi uwsgi
RUN pip install Flask==0.10.1 uWSGI==2.0.8 requests==2.5.1 redis==2.10.3 ❶
WORKDIR /app
COPY app /app
COPY cmd.sh /

EXPOSE 9090 9191
USER uwsgi

CMD ["/cmd.sh"]
```

❶ 我们只需安装 Python 的 Redis 客户端库。

这是更新后的 docker-compose.yml：

```
identidock:
  build: .
  ports:
   - "5000:5000"
  environment:
    ENV: DEV
  volumes:
    - ./app:/app
  links:
    - dnmonster
    - redis  ❶

dnmonster:
  image: amouat/dnmonster:1.0

redis:
  image: redis:3.0 ❷
```

❶ 建立与 Redis 容器的连接。
❷ 创建一个基于官方 Redis 镜像的容器。

现在，如果你先用 `docker-compose stop` 停止 identidock，可以用 `docker-compose build` 和 `docker-compose up` 来启动新版本。因为我们并没有在功能上做任何改变，所以你应该不会发现新版本有什么不一样的地方。如果你想确认新代码真的已经在执行，那么可以查看调试信息；或者，如果你还是不太放心的话，试试接上一个监控工具，例如第 10 章将会介绍的 Prometheus，看看当你给应用程序产生负载时会发生什么事情。

6.4 微服务

我们已经根据微服务架构开发了 identidock 应用程序，因为这个系统是由多个独立的小服务所组成的。这种风格往往被拿来和单一服务架构比较，单一服务架构指的是该系统包含在一个单独的大型的服务之中。尽管 identidock 是一个比较儿戏的应用程序，它毕竟还是凸显了微服务风格的一些特征。

假如采用单一服务架构实现的话，那就等于 dnmonster、Redis 以及 identidock 全都使用同一种编程语言实现，并且它们必须融为一体，在同一个容器中运行。如果设计得宜，那么一整个程序中的不同部件可以被拆分成单独的库，并尽可能使用现有的库。

相比之下，我们的 identidock 应用由三个容器组成，其中有一个用 Python 写成的 Web 程序，它与一个用 JavaScript 写成的服务，以及一个用 C 语言实现的键值数据库通信。后面的章节中将介绍如何用少量的工作来使更多的微服务接入 identidock，包括第 9 章的反向代理，以及第 10 章的监控和日志记录方案。

采取这种方法有几个优点。微服务架构的系统很适合横向扩展到多台机器。微服务能够轻松快速地被其他效能更高且功能相同的服务替代。假如发生意外情况，可以只对部分微服务进行回滚，而无需把系统的其他部分一并关闭。不同的微服务可以用不同的语言实现，使开发者能够选择适用于手头任务的语言。

但微服务也有不足之处，主要在于分布式组件所导致的额外开销。它们之间的通信必须通过网络，而不是库的调用。我们必须使用 Compose 这样的工具，以确保所有组件同时启动并连接正确。服务编排和服务发现成为亟需解决的重要问题。

微服务能够提供更高的扩展能力以及灵活性，使得新式的互联网应用从中受益良多，这一点已经在 Netflix、亚马逊和 SoundCloud 等公司身上得以证明。鉴于此，微服务将会是一个意义重大且影响巨大的架构，不过和众多技术一样，它也并非万能。[4]

6.5 总结

现在，我们的应用已经基本可用了。虽然它还很简单，但已具备足够多的功能，能以多个容器的方式实现，并突出强调了在开发中使用容器时需要注意的事项。我们已经了解了如何重复使用现有的镜像，无论是作为构建应用的基础（例如我们使用的 Python 基础镜像），还是像黑盒般只用于提供某个服务（例如 dnmonster 镜像）。

最重要的是，我们还看到了容器如何自然演变成一组具有明确功能的小服务，它们可以交互形成一个更大的系统。这就是微服务架构。

注 4：更多关于微服务优缺点的信息，可以参考 Martin Fowler 的文章，其中包括“Microservices”（http://martinfowler.com/articles/microservices.html）。

第 7 章

镜像分发

一旦创建好了镜像，你应该希望其他人也能够使用，无论是分享给你的同事、持续集成的服务器，还是最终用户。有几种方法可以用来分发镜像：从 Dockerfile 重新构建、从寄存服务器下载，或通过 `docker load` 命令从归档文件安装。

本章将深入探讨这些方法之间的差异，以及研究为团队内和团队外的用户分发镜像的最佳方案。我们将看到如何给 identidock 镜像加标签并将其上传，使得它可以在工作流程的其他部分中使用，并提供给他人下载。

> 本章的代码见于本书的 GitHub 仓库（https://github.com/using-docker/image-dist）。`v0` 标签代表代码在上一章结尾时的版本；当我们不断往前推进时，标签也会不断更新。用以下命令获取这个版本的代码：
>
> ```
> $ git clone -b v0 \
> https://github.com/using-docker/image-dist/
> ...
> ```
>
> 除此以外，还可以在 GitHub 项目的 Releases 页面（https://github.com/using-docker/image-dist/releases）下载各个标签的代码。

7.1　镜像及镜像库的命名方式

之前的 3.4 节中已经介绍了如何给镜像指定合适的标签，并把它们上传到远程仓库。分发镜像的时候使用能精确描述镜像的名称和标签是非常重要的。下面来重温一下，指定镜像的名称和标签有两种方法，一是在构建镜像的时候，二是通过 `docker tag` 命令：

```
$ cd identidock
$ docker build -t "identidock:0.1" . ❶
$ docker tag "identidock:0.1" "amouat/identidock:0.1" ❷
```

❶ 指定仓库名称为 `identidock`，标签为 `0.1`。

❷ 把 `amouat/identidock` 名称与镜像关联，这一名称指镜像属于 Docker Hub 上用户名为 amouat 的用户。

> **当心 `latest` 标签**
>
> 别让 `latest` 标签误导你！当没有指定标签的时候，Docker 会使用这个标签作为默认值，但除此之外，它不具备任何特殊含义。很多仓库都会把它作为最新的稳定版镜像的别名，但这只不过是一个惯例，而并非出于任何严格规定。与所有其他镜像一样，即使有新版本被推送到寄存服务器，`latest` 标签的镜像也不会自动更新——你仍需明确执行 `docker pull` 命令来获取最新版本。
>
> 当执行 `docker run` 或 `docker pull` 时，如果指定的镜像名称不带标签，那么 Docker 就会使用带 `latest` 标签的镜像，如果的确存在这样一个镜像；否则就会报错。
>
> 由于很多用户弄不明白 `latest` 标签，或许你应该考虑干脆不使用它，尤其是在面向公众的仓库中。

标签名称必须遵循一些规则，必须由大写或小写字母、数字，以及 . 和 - 符号组成。它们的长度必须为 1~128 个字符。第一个字符不能是 . 或 - 符号。

建立开发工作流程时，仓库和标签的命名是非常重要的。对于正确的命名，Docker 的限制非常少，而且允许随时创建和删除名称。这意味着建立并实施可行的命名方案的主动权掌握在开发团队手中。

7.2 Docker Hub

让你的镜像能够供别人使用，最直接的方法就是使用 Docker Hub。Docker Hub 是 Docker 公司提供的一个线上镜像寄存服务。对于公开的镜像，Docker Hub 提供免费的仓库，用户也可以付费获得私有仓库。

> **其他私有托管服务**
>
> Docker Hub 并非唯一的能在网上提供私有仓库的地方。本书写作之际，quay.io 是这里的一个主要的竞争者，它以具有竞争优势的价格提供比 Docker Hub 还要多的功能。

上传 identidock 镜像非常容易。假设已经有了 Docker Hub 的账户，[1] 可以直接在命令行中执行以下操作：

注 1：如果还没有账号，你可以在 https://hub.docker.com 注册一个。

```
$ docker tag identidock:latest amouat/identidock:0.1 ❶
$ docker push amouat/identidock:0.1 ❷
The push refers to a repository [docker.io/amouat/identidock] (len: 1)
76899e56d187: Image successfully pushed
...
0.1: digest: sha256:8aecd14cb97cc4333fdffe903aec1435a1883a44ea9f25b45513d4c2...
```

❶ 我们需要做的第一件事便是在 Docker Hub 的用户命名空间中为镜像创建一个别名。换句话说，别名的形式必须是 `<username>/<repositoryname>`，其中的 `<username>` 是你在 Docker Hub 上的用户名（例如 `amouat` 是我的用户名），而 `<repositoryname>` 是你希望在 Docker Hub 上使用的仓库名称。我们也借此机会把镜像的标签设置为 `0.1`。

❷ 用我们刚刚创建的别名把镜像推送到 Docker Hub。如果仓库不存在，它会创建一个新的，并且将镜像上传到适当的标签下。

至此，identidock 镜像已经公开了，任何人都可以通过执行 `docker pull` 命令来获取它。

如果现在访问 Docker Hub 网站，就能在类似 https://registry.hub.docker.com/u/amouat/identidock/ 的 URL 下找到你的仓库。登录之后还能执行各种管理任务，譬如设置仓库的描述、把其他用户加入为合作者，以及建立 webhook。

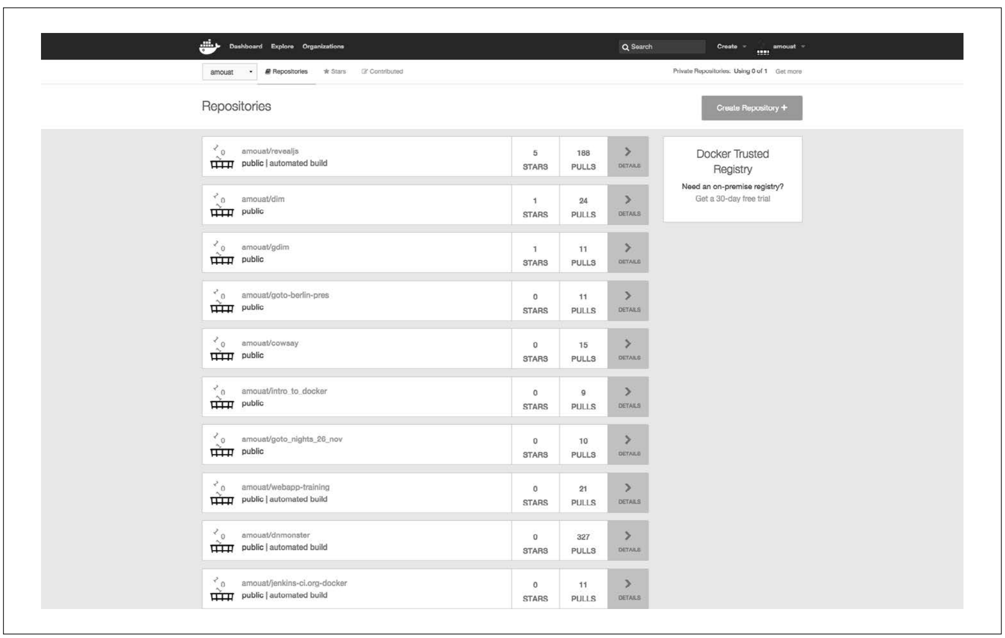


图 7-1：Docker Hub 首页

每当需要更新仓库时，我们只需对更新的镜像重复标签和推送的步骤。如果使用现有的标签，那么先前的镜像将被覆盖。虽然这样很不错，但如果我们希望每当代码有更新的时候，镜像也能随之更新，那又该怎么办呢？ 这其实是一个非常普遍的用例；出于这个原因，Docker Hub 引入了自动构建的概念。

7.3 自动构建

让我们在 Docker Hub 上为 identidock 配置自动构建。一旦自动构建成功后，每当推送任何代码修改，Docker Hub 就会构建 identidock 镜像，并将其保存到我们的仓库。要做到这一点，你需要建立一个 GitHub 或 Bitbucket 的仓库。你可以把目前已有的代码推送上去，或把官方的代码“fork”[2] 一份出来，官方代码可以在本书的 GitHub 项目（https://github.com/using-docker/image-dist）中找到。

自动构建是通过 Docker Hub 的网页界面进行配置的，而不是通过命令行。如果你已经登录网站，你应该在右上角看到一个名为“Create”的下拉菜单。在这里，选择“Create Automated Build”，并找出 identidock 代码的仓库。[3] 选择仓库后，会跳转到自动构建的配置页面。仓库名称默认为代码库，你应该把名称改为一个有意义的名字，例如 `identidock_auto`。给仓库加一个简短的说明，如“自动构建的 identidock 镜像”。第一个“Tag”字段应为 `Branch`，而它的名称为 `master`，意思是跟踪主分支（master）的代码。如果你是从我的仓库把代码 fork 出来的，那么你应该把“Dockerfile Location”设置为 /identidock/Dockerfile。最后的“Tag”字段决定分配给 Docker Hub 上的镜像名称。你可以使用默认的 `latest`，或改用一个更有意义的名称，如 `auto`。当完成了所有步骤之后，点击“Create”。Docker 跳转到这个新仓库的构建页面。现在可以点击“Trigger a Build”来生成你的第一个自动构建镜像。当完成构建后，你便可以下载镜像了（假设生成成功）。

我们可以把源代码稍微改动一下，以测试自动构建是否已生效。我们将要添加一个 README 文件，Docker Hub 会用它来展示一些仓库信息。在 identidock 目录下创建一个含有项目简短描述的 README.md 文件，内容如下：[4]

```
identidock
==========

Simple identicon server based on monsterid from Kevin Gaudin.

From "Using Docker" by Adrian Mouat published by O'Reilly Media.
```

提交这个文件，并把它推送到 Github：

```
$ git add README.md
$ git commit -m "Added README"
[master d8f3317] Added README
 1 file changed, 6 insertions(+)
 create mode 100644 identidock/README.md
$ git push
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
```

注 2：fork 一词有分支或衍生的意思，为软件工程术语，常用于开源软件项目，并为 Github 的主要功能之一。在 Github 上对某个软件的源代码仓库进行 fork 的操作后，Github 便会把该仓库复制一份到用户自己的账号下，用户可以对其进行任何修改而不会影响原项目。——译者注

注 3：你必须先关联你的 GitHub 或 Bitbucket 账户，如果你还没有这样做的话。

注 4：如果你从我的仓库把代码 fork 出来，这个文件就已经存在了，你随便更改其中一些文字就行。

```
Writing objects: 100% (4/4), 456 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
To git@github.com:using-docker/image-dist.git
   c81ff68..d8f3317 master -> master
```

如果稍等片刻，然后再访问仓库的构建页面，应该会看到它正在构建一个新版本的镜像。

无论何种原因，如果构建失败了，你可以点击“Build Details”标签页中的“Build Code”来查看日志。你也可以随时点击“Trigger a Build”按钮来生成一个新的镜像。

这种构建和分发镜像的方式并非适用于所有项目。除非你已付费使用私有仓库，否则你的镜像是公开的，而且你还得完全依赖 Docker Hub——要是 Docker Hub 出现宕机，你就无法更新镜像，用户也将无法下载它们。还要考虑效率：如果你需要快速构建和在不同机器之间传送镜像，那么从 Docker Hub 下载镜像和排队等待镜像生成的这些开销绝对不是你想要的。对于开源项目和小型项目，Docker Hub 堪称完美。但对于一些更大型或更严谨的项目，最好能找到其他方案进行替换或补充。

7.4　私有分发

Docker Hub 之外有几个选项可供选择。你可以手动处理，要么导出然后导入镜像，要么简单地在每一个 Docker 主机上利用 Dockerfile 重新构建镜像。但这两种方案都不太理想：每次从 Dockerfile 构建镜像都会很慢，并且可能导致不同主机上镜像有所差异；导出和导入镜像处理起来必须很小心，而且容易出错。剩下的选择就只有使用别的寄存服务，可以是自己维护的，或由第三方托管。

下面先来看一下免费的解决方案，即运行你自己的寄存服务，之后再来看有哪些商业产品可以考虑。

7.4.1　运行自己的寄存服务

Docker 的寄存服务与 Docker Hub 并不一样。虽然两者都实现了寄存服务的 API，可以让用户推送、下载和搜索镜像，但 Docker Hub 是一个闭源的远程服务，而寄存服务则是能够在本地运行的一个开源应用程序。Docker Hub 还包括用户的账户管理、统计数据和网页界面，这些都是寄存服务没有提供的。

开发进行中

虽然寄存服务 v2 是稳定版，但有若干个重要的功能仍在开发中。鉴于此，我会在本节集中说明一些常用的用例，而省略那些高级功能的细节。关于寄存服务的完整且最新的文档可以在 Docker 的 GitHub 项目（https://github.com/docker/distribution）中找到 。

本章中只会使用寄存服务的版本 2，而它只支持 Docker 守护进程的 1.6 或更高版本。如果需要支持旧版本的 Docker，你需要运行寄存服务的上一个版本（在版本迁移阶段，也可以同时运行两个版本的寄存服务）。对比版本 1，版本 2 的安全性、可靠性和效率都更高，因此我强烈建议你在条件允许的情况下尽量使用版本 2。

本地运行寄存服务的最简单方法就是使用官方镜像。这样我们便可以快速运行起来：

```
$ docker run -d -p 5000:5000 registry:2
...
75fafd23711482bbee7be50b304b795a40b7b0858064473b88e3ddcae3847c37
```

现在我们有了一个运行中的寄存服务，可以用这个服务来给镜像加标签并推送给它。如果你使用的是 `docker-machine`，你也可以使用 `localhost` 地址，因为解析这个地址的是与寄存服务运行在同一主机上的 Docker 引擎（而不是客户端）：

```
$ docker tag amouat/identidock:0.1 localhost:5000/identidock:0.1
$ docker push localhost:5000/identidock:0.1
The push refers to a repository [localhost:5000/identidock] (len: 1)
...
0.1: digest: sha256:d20affe522a3c6ef1f8293de69fea5a8621d695619955262f3fc2885...
```

如果现在把本地的版本删掉，还可以再次下载：

```
$ docker rmi localhost:5000/identidock:0.1
Untagged: localhost:5000/identidock:0.1
$ docker pull localhost:5000/identidock:0.1
0.1: Pulling from identidock
...
76899e56d187: Already exists
Digest:sha256:d20affe522a3c6ef1f8293de69fea5a8621d695619955262f3fc28852e173108
Status: Downloaded newer image for localhost:5000/identidock:0.1
```

Docker 发现已有内容相同的镜像，因此实际上发生的这一切只是标签被加回去而已。可能你已注意到，寄存服务为镜像生成了一个*摘要值*（digest）。这是基于镜像和它的元数据产生的一个唯一的散列值。你可以利用这个值来下载镜像，像这样：

```
$ docker pull localhost:5000/identidock@sha256:\
d20affe522a3c6ef1f8293de69fea5a8621d695619955262f3fc28852e173108
sha256:d20affe522a3c6ef1f8293de69fea5a8621d695619955262f3fc28852e173108: Pul...
...
76899e56d187: Already exists
Digest: sha256:d20affe522a3c6ef1f8293de69fea5a8621d695619955262f3fc28852e173108
Status: Downloaded newer image for localhost:5000/identidock@sha256:d20affe5...
```

使用摘要的主要优点是，它能保证下载的镜像确实是你想要的。如果使用标签下载，镜像有可能在你不知情的时候已经被更改过。此外，使用摘要保证了镜像的完整性；你可以肯定它在传输的过程当中或存储时没有被篡改过。关于如何安全处理镜像以及建立它们的出处，参见 13.6 节。

需要寄存服务的主要原因是，它能作为你的团队或组织的中央存储。这意味着你需要在远程的 Docker 守护进程中也能从寄存服务器下载镜像。但是，如果尝试对我们的寄存服务器这样做，将会发生以下错误：

```
$ docker pull 192.168.1.100:5000/identidock:0.1 ❶
Error response from daemon: unable to ping registry endpoint
https://192.168.99.100:5000/v0/
v2 ping attempt failed with error: Get https://192.168.99.100:5000/v2/:
tls: oversized record received with length 20527
```

```
 v1 ping attempt failed with error: Get https://192.168.99.100:5000/v1/_ping:
tls: oversized record received with length 20527
```

❶ 我在这里以服务器的 IP 地址取代了“localhost”。无论你执行下载动作的守护进程是在另一台计算机上，还是与寄存服务处于同一台计算机上，都会发生这一错误。

那么，到底发生了什么呢？原来 Docker 守护程序拒绝连接到远程主机，是因为它没有一个有效的传输层安全（Transport Layer Security，TLS）证书。之前能够工作，只是因为 Docker 特别允许从“localhost”服务器下载。修复这个问题的方法有 3 种。

(1) 对将要访问寄存服务器的所有 Docker 守护进程加上 `--insecure-registry 192.168.1.100:5000` 参数，其中的地址和端口需要替换成你的服务器的信息，然后重新启动 Docker 守护进程。
(2) 在主机上安装一个来自可信的证书颁发机构签署的证书，如同需要为一个可以通过 HTTPS 访问的网站所做的那样。
(3) 在主机上安装一个自签名的证书（self-signed certificate），并同时给需要访问寄存服务器的每个 Docker 守护进程都安装一份。

第一个方法是最简单的，但出于安全，我们不会考虑它。第二个方法是最好的，但需要一个来自可信的证书认证机构签发的证书，而这通常都有相关的费用。第三个方法是安全的，但需要将证书手动复制到每个守护进程。

如果想创建一张你自己的自签名证书，可以使用 OpenSSL 工具。这些步骤需要在一台打算长期用作寄存服务器的计算机上进行。这些步骤在 Digital Ocean 托管的 Ubuntu 14.04 虚拟机上测试过；在其他操作系统上执行可能会有差异。

```
root@reginald:~# mkdir registry_certs
root@reginald:~# openssl req -newkey rsa:4096 -nodes -sha256 \
>     -keyout registry_certs/domain.key -x509 -days 365 \
      -out registry_certs/domain.crt ❶
Generating a 4096 bit RSA private key
....................................................++
....................................................++
writing new private key to 'registry_certs/domain.key'
-----
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:reginald ❷
Email Address []:
root@reginald:~# ls registry_certs/
domain.crt domain.key ❸
```

❶ 创建一个 x509 的自签名证书和 4096 位的 RSA 私钥。这张证书使用 SHA256 的摘要签署，其有效期为 365 天。OpenSSL 还会要求你提供一些信息，你可以输入新的值或选择保留默认值。

❷ common name 很重要；它必须与用来访问服务器的名字相同，而且不可以使用 IP 地址（“reginald”是我的服务器的名字）。

❸ 当这个过程结束后，我们会有一张名为 domain.crt 的证书，该证书将会与各个客户端共用。还有一把名为 `domain.key` 的密钥，密钥的储存必须保证安全，永远不能与别人共享。

通过 IP 地址访问寄存服务

如果想使用 IP 地址来访问寄存服务，配置就会变得复杂。你不能简单地把 IP 地址用作 common name。你必须为你打算使用的各个 IP 地址设置 Subject Alternative Name（SAN）。

一般情况下，我不赞成这样做。为你的服务器挑选一个名字，并让它能在内网中被寻址，肯定是个比较好的做法（即使在最坏的情况下，也可以手动把服务器名称添加到 /etc/hosts 中）。这个做法通常更容易配置，并且即使 IP 地址更改了，也不需要把所有镜像重新赋予标签。

接下来需要将证书复制到每一个将会访问寄存服务器的 Docker 守护进程。[5] 复制的目标文件是 /etc/docker/certs.d/<registry_address>/ca.crt，其中的 <registry_address> 为你的寄存服务器的地址和端口。你还需要重新启动 Docker 守护进程。举例如下：

```
root@reginald:~# sudo mkdir -p /etc/docker/certs.d/reginald:5000
root@reginald:~# sudo cp registry_certs/domain.crt \
                 /etc/docker/certs.d/reginald:5000/ca.crt  ❶
root@reginald:~# sudo service docker restart
docker stop/waiting
docker start/running, process 3906
```

❶ 如果需要在远程的主机上执行，你需要利用 scp 或类似的工具，将 CA 证书传到该 Docker 主机。假如你使用了一个公共可信的证书认证机构签发的证书，便可以跳过这一步。

现在可以启动寄存服务：[6]

```
root@reginald:~# docker run -d -p 5000:5000 \
             -v $(pwd)/registry_certs:/certs \ ❶
             -e REGISTRY_HTTP_TLS_CERTIFICATE=/certs/domain.crt \
             -e REGISTRY_HTTP_TLS_KEY=/certs/domain.key \  ❷
             --restart=always --name registry registry:2
...
b79cb734d8778c0e36934514c0a1ed13d42c342c7b8d7d4d75f84497cc6f45f4
```

❶ 以数据卷的方式把证书放入容器。

注 5：如果你有一张由一家可信的认证中心签发的证书，可以跳过这一步。

注 6：你可能需要删除先前曾经启动过的寄存服务实例。

❷ 可以通过环境变量配置寄存服务来使用我们的证书。为了证明一切能正常运作，试试下载镜像，把它重新加标签，并把它推送出去：

```
root@reginald:~# docker pull debian:wheezy
wheezy: Pulling from library/debian
ba249489d0b6: Pull complete
19de96c112fc: Pull complete
library/debian:wheezy: The image you are pulling has been verified.
Important: image verification is a tech preview feature and should not be
relied on to provide security.
Digest: sha256:90de9d4ecb9c954bdacd9fbcc58b431864e8023e42f8cc21782f2107054344e1
Status: Downloaded newer image for debian:wheezy
root@reginald:~# docker tag debian:wheezy reginald:5000/debian:local ❶
root@reginald:~# docker push reginald:5000/debian:local
The push refers to a repository [reginald:5000/debian] (len: 1)
19de96c112fc: Image successfully pushed
ba249489d0b6: Image successfully pushed
local: digest: sha256:3569aa2244f895ee6be52ed5339bc83e19fafd713fb1138007b987...
```

❶ 必须把“reginald”替换成你的服务器名称。

我们终于有了一个能安全地存储镜像的远程寄存服务。当你从其他机器进行测试时，谨记将证书文件复制到 Docker 引擎主机上的 /etc/docker/certs.d/<registry_address>/ca.crt，并确保 Docker 引擎能够解析寄存服务器的地址。[7]

Docker 拥有很多配置选项，针对不同的使用情况，可以利用它们来设置和调整寄存服务的行为。寄存服务的选项由镜像中的一个 YAML 文件配置，可以用数据卷将其替换。执行程序时，选项的值也可以通过环境变量来重新定义，例如前面例子中的 `REGISTRY_HTTP_TLS_KEY` 和 `REGISTRY_HTTP_TLS_CERTIFICATE`。本书写作时，配置文件位于 /go/src/github.com/docker/distribution/cmd/registry/config.yml，但将来这个路径很可能会简化。默认的配置内容专为开发环境而设，如需用于生产环境，那么必须对它进行大幅度修改。可以在 Docker 的 GitHub 项目找到如何配置寄存服务的所有细节，以及配置文件的范例。

余下的内容将讲述建立一个寄存服务时需要考虑的主要功能以及定制的功能。

1. 存储

寄存服务器的镜像默认使用文件系统驱动，顾名思义，所有数据和镜像将会保存在文件系统之上。对于开发环境，甚至很多应用场景而言，这是个很好的选择。你需要在已定义的根目录使用一个数据卷，并把它指向一个可靠的文件存储。例如，以下的 config.yml 将配置寄存服务使用文件系统驱动，并将数据保存于 /var/lib/registry，而且要把它定义为一个数据卷：

```
storage:
    filesystem:
        rootdirectory: /var/lib/registry
```

如果要把数据保存到云中，可以使用亚马逊 S3 和微软 Azure 存储驱动。

注 7：不能把寄存服务的名称替换成 IP 地址，因为它将无法与证书匹配。相反，应该编辑 /etc/hosts 文件或配置 DNS，使域名能够被解析。

此外，它还支持 Ceph 分布式对象存储，并利用 Redis 作为内存缓存来加速镜像层的访问效率。

2. 身份验证

目前为止，我们已经知道如何使用 TLS 访问寄存服务，但尚未涉及任何用户身份验证的相关内容。如果只使用公共镜像，或寄存服务只能在内网访问，这样或许无伤大雅，但大多数机构还是希望只有通过验证的用户才有访问权限。

有两种实现方法。

(1) 在寄存服务之前设置一个代理（如 nginx）负责验证用户。GitHub 项目的官方文档（https://docs.docker.com/registry/recipes/nginx/）中提供了一个使用 nginx 用户名 / 密码作为认证的范例 。一旦配置成功，`docker login` 命令便可用于寄存服务的身份验证。

(2) 使用基于 JSON 网络令牌（JSON Web Token）实现的令牌认证。使用此方法后，无法出示有效令牌的客户端会被寄存服务器拒绝访问，并将其重定向到身份认证服务器。客户端可以从认证服务器取得令牌，然后以此访问寄存服务。Docker 尚未提供认证服务器，撰写本书的时候只有一个由 Cesanta Software（https://github.com/cesanta/docker_auth）提供的开源解决方案。目前而言，其他选择只有自己开发一套基于 JSON 网络令牌库的方案，或付费使用 7.4.2 节中提及的商用解决方案。虽然这样做显然更加复杂和困难，但对于许多大型或架构分散的机构而言却非常重要。

3. HTTP

这里将会配置寄存服务的 HTTP 接口。你必须确保它的配置正确无误，它才能正常工作。特别要注意的是，它的 TLS 证书和密钥的位置必须配置好；在前面的例子中，我们是通过 `REGISTRY_HTTP_TLS_KEY` 和 `REGISTRY_HTTP_TLS_CERTIFICATE` 这两个环境变量来做的。

通常的配置如下所示：

```
http:
    addr: reginald:5000 ❶
    secret: DD100CC4-1356-11E5-A926-33C19330F945 ❷
    tls: ❸
        certificate: /certs/domain.crt
        key: /certs/domain.key
```

❶ 寄存服务器的地址。

❷ 用来签署客户端存储的状态信息的一个随机字符串。它的目的是为了防止信息被篡改。理想的情况下这个字符串应该是随机生成的。

❸ 就像之前看到的那样，把证书设置妥当。这些文件必须让容器能访问得到，可以通过挂载数据卷，或把文件复制到容器中。

4. 其他配置

请注意，还有其他选项用于配置中间件、通知、日志记录和缓存。欲了解更多详情，请参阅 Docker 的 GitHub 项目。

7.4.2　商业寄存服务

如果你正在寻找一套更完整的基于 Web 的管理解决方案，可以选择 Docker Trusted Registry（https://www.docker.com/docker-trusted-registry）或 CoreOS Enterprise Registry（https://coreos.com/products/enterprise-registry/）。这两套产品都是用于企业内部的商业解决方案，因此受到企业的防火墙保护。

它们不仅能够存储镜像，还具备很多其他功能。它们都提供了针对团队工作环境的 Docker 镜像管理工具，例如细粒度的权限控制，以及处理安装和管理任务的图形用户界面。

7.5　缩减镜像大小

现在你可能已经注意到，Docker 镜像都比较大；大多数的镜像的大小都有上百 MB，意味着大量的时间都花费在等待镜像的来回传送上。镜像的层次结构可以缓解这一问题；如果已经有了一个镜像的父层，只需下载新的子镜像层就可以了。

但是，关于如何缩减镜像的大小仍有很多讨论空间，只不过说易行难。一个单纯的想法是从镜像中删除不需要的文件。不幸的是，这是行不通的。记住，镜像是由多个层所组成的，每个镜像对应 Dockerfile 以及其上的所有 Dockerfile 的每一个命令。镜像的总大小是所有镜像层的总和。即使在一个镜像层中删除了一个文件，它也仍然存在于父镜像层。具体的例子请看以下的 Dockerfile：

```
FROM debian:wheezy

RUN dd if=/dev/zero of=/bigfile count=1 bs=50MB ❶
RUN rm /bigfile
```

❶ 这个命令只是用来快速创建一个文件。

如果现在构建并检查这个镜像：

```
$ docker build -t filetest .
...
$ docker images filetest ❶
REPOSITORY  TAG     IMAGE ID      CREATED        VIRTUAL SIZE
filetest    latest  e2a98279a101  8 seconds ago  135 MB
$ docker history filetest ❷
IMAGE         ... CREATED BY                                  SIZE     ...
e2a98279a101      /bin/sh -c rm /bigfile                      0 B
5d0f04380012      /bin/sh -c dd if=/dev/zero of=/bigfile count=   50 MB
c90d655b99b2      /bin/sh -c #(nop) CMD [/bin/bash]           0 B
30d39e59ffe2      /bin/sh -c #(nop) ADD file:3f1a40df75bc5673ce   85.01 MB
511136ea3c5a                                                  0 B
```

❶ 这里可以看到，镜像的总大小为 135MB，正好比基础镜像大 50MB。

❷ `docker history` 给出了完整的信息。最上面的两行描述我们的 Dockerfile 所创建的镜像层。可以看到，`dd` 命令创建了一个大小为 50MB 的镜像层，而 `rm` 命令在它上面又新建了一个层。

相反，如果我们的 Dockerfile 是这样的：

```
FROM debian:wheezy

RUN dd if=/dev/zero of=/bigfile count=1 bs=50MB && rm /bigfile
```

我们再来执行构建和检查它：

```
$ docker build -t filetest .
...
$ docker images filetest
REPOSITORY  TAG     IMAGE ID      CREATED          VIRTUAL SIZE
filetest    latest  40a9350a4fa2  34 seconds ago   85.01 MB
$ docker history filetest
IMAGE        ... CREATED BY                                  SIZE     ...
40a9350a4fa2     /bin/sh -c dd if=/dev/zero of=/bigfile count=   0 B
c90d655b99b2     /bin/sh -c #(nop) CMD [/bin/bash]               0 B
30d39e59ffe2     /bin/sh -c #(nop) ADD file:3f1a40df75bc5673ce   85.01 MB
511136ea3c5a                                                     0 B
```

我们并没有增加基础镜像的大小。如果在同一个镜像层中创建文件后再把它删除，那么文件是不会被包含在镜像中的。正因为如此，你将会经常碰见一些 Dockerfile，它会在同一个 RUN 指令中完成下载压缩包或其他归档文件、将其解压，以及立即删除归档文件这几个动作。例如，官方的 MongoDB 镜像包括以下指令（为便于排版，URL 已缩短）：

```
RUN curl -SL "https://$MONGO_VERSION.tgz" -o mongo.tgz \
    && curl -SL "https://$MONGO_VERSION.tgz.sig" -o mongo.tgz.sig \
    && gpg --verify mongo.tgz.sig \
    && tar -xvf mongo.tgz -C /usr/local --strip-components=1 \
    && rm mongo.tgz*
```

类似的策略还可以应用于源码——有时候你会发现同一行内可以完成代码下载、二进制编译，以及文件的删除。

同理，试图清理包管理工具是没有任何意义的：

```
RUN rm -rf /var/lib/apt/lists/*
```

但是可以这样做（同样摘自官方的 mongo Dockerfile）：

```
RUN apt-get update \
    && apt-get install -y curl numactl \
    && rm -rf /var/lib/apt/lists/*
```

另外，请参阅 4.2.4 节中关于如何明智地选择基础镜像以尽可能缩减镜像大小的讨论。

如果遇上不得不缩减镜像大小的紧要关头，还有另外一个选择。先对一个容器执行 docker export，再对其结果执行 docker import，就会得到一个只含有一个层的镜像。举例如下：

```
$ docker create identidock:latest
fe165be64117612c94160c6a194a0d8791f4c6cb30702a61d4b3ac1d9271e3bf
$ docker export $(docker ps -lq) | docker import -
146880a742cbd0e92cd9a79f75a281f0fed46f6b5ece0219f5e1594ff8c18302
$ docker tag 146880a identidock:import
$ docker images identidock
REPOSITORY  TAG     IMAGE ID      CREATED        VIRTUAL SIZE
identidock  import  146880a742cb  5 minutes ago  730.9 MB
```

```
identidock  0.1     76899e56d187  23 hours ago   839.5 MB
identidock  latest  1432cc6c20e5  4 days ago     839 MB
$ docker history identidock:import
IMAGE         CREATED          CREATED BY  SIZE       COMMENT
146880a742cb  11 minutes ago               730.9 MB   Imported from -
```

这样镜像就变得更小了，不过会付出以下代价。

- 需要重新处理所有未反映在文件系统里的 Dockerfile 指令，如 EXPOSE、CMD 和 PORTS。
- 与镜像关联的所有元数据将会丢失。
- 再也不能与其他具有相同父层的镜像共享空间。

7.6　镜像出处

分发和使用镜像时，必须考虑如何确立它们的出处（provenance），也就是镜像来自何处以及来自谁。下载镜像的时候，需要确保它是真正由它所声称的作者创建的，而且没有被篡改过，并且它与镜像作者测试的镜像是同一个。

为了满足这些需求，Docker 推出了名为内容信任（Docker content trust，https://docs.docker.com/engine/security/trust/content_trust/）的解决方案，撰写本书的时候，它还在测试阶段，默认未启用。更多详情参见 13.6 节。

7.7　总结

在采用 Docker 的工作流程当中，有效的镜像发布是关键的组成部分。本章探讨了主要的解决方案：Docker Hub 和私有的寄存服务。我们还学习了一些分发镜像时需要注意的事项，包括需要给镜像设置合适的名称和标签，以及如何缩减镜像的大小。

下一章将会把镜像带进工作流的下一步，即持续集成服务。

第 8 章

Docker持续集成与测试

本章将介绍如何利用 Docker 和 Jenkins 创建一个持续集成（continuous integration，CI）的工作流来构建和测试我们的应用程序。本章同时也会涉及 Docker 测试的其他方面，还会对测试微服务架构作简单介绍。

容器和微服务的测试各有各的挑战。微服务使单元测试变得容易，但由于服务和网络连接的数量有所增加，使得系统和集成测试变得困难。这时候，模拟网络服务比模拟单一架构系统中传统的 Java 或 C# 类更有意义。把测试代码保留在镜像中，好处是能保证容器的可移植性和一致性，坏处是增加了它们的大小。

本章代码见于本书的 GitHub 仓库（https://github.com/using-docker/ci-testing）。标签 `v0` 代表 identidock 代码在上一章为止的状态，随着代码不断往前发展，代码的标签也会同步推进。用以下命令来获取这个版本的代码：

```
$ git clone -b v0 \
https://github.com/using-docker/ci-testing/
...
```

除此之外，也可以在 GitHub 项目的 Releases 页面（https://github.com/using-docker/ci-testing/releases）下载各个标签的代码。

8.1　为identidock添加单元测试

首先应该添加一些单元测试到我们的 identidock 代码里。在无需依赖于任何外部服务的情况下，这些单元测试将对 identidock 代码进行一些基本功能的测试。[1]

注 1：许多开发者提倡测试驱动开发（test-driven development，TDD）模式，实施这种模式时会先把测试写好，然后才完成能通过这些测试的代码。本书并没有遵循这种开发模式，主要是为了方便叙述。

用以下内容来创建一个 identidock/app/tests.py 文件：

```
import unittest
import identidock

class TestCase(unittest.TestCase):

    def setUp(self):
        identidock.app.config["TESTING"] = True
        self.app = identidock.app.test_client()

    def test_get_mainpage(self):
        page = self.app.post("/", data=dict(name="Moby Dock"))
        assert page.status_code == 200
        assert 'Hello' in str(page.data)
        assert 'Moby Dock' in str(page.data)

    def test_html_escaping(self):
        page = self.app.post("/", data=dict(name='"><b>TEST</b><!--'))
        assert '<b>' not in str(page.data)


if __name__ == '__main__':
    unittest.main()
```

这是一个非常简单的测试文件，其中只有 3 个方法。

`setUp`

基于我们的 Flask Web 应用，初始化一个它的测试版本。

`test_get_mainpage`

这个测试方法以“Moby Dock”作为 `name` 字段的输入并调用 URL `/`。然后，它会检查方法是否返回 200 状态码，以及返回的数据是否包含“Hello”和“Moby Dock”字符串。

`test_html_escaping`

测试输入中的 HTML 元素能否被正确转义。

下面来运行这些测试：

```
$ docker build -t identidock .
...
$ docker run identidock python tests.py
.F
======================================================================
FAIL: test_html_escaping (__main__.TestCase)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "tests.py", line 19, in test_html_escaping
    assert '<b>' not in str(page.data)
AssertionError

----------------------------------------------------------------------
Ran 2 tests in 0.010s

FAILED (failures=1)
```

嗯，看起来不太对劲儿。第一个测试顺利通过，但第二个测试却失败了，因为我们没有将用户的输入正确地转义。这是一个严重的安全问题，在大型应用中可能导致数据泄漏和跨站脚本攻击（XSS）。如果你想看看对程序有什么影响，可以启动 identidock 并尝试对 name 字段输入类似 "> pwned!<!--"，必须包括引号。攻击者有可能向我们的应用程序注入恶意的 JavaScript 代码，并诱使用户执行它。

但值得庆幸的是，这个问题很容易修复。只需更改我们的 Python 程序，让它把用户的输入进行净化，使 HTML 元素和引号被转义成适当的代码。把 identidock.py 的代码更新如下：

```
from flask import Flask, Response, request
import requests
import hashlib
import redis
import html

app = Flask(__name__)
cache = redis.StrictRedis(host='redis', port=6379, db=0)
salt = "UNIQUE_SALT"
default_name = 'Joe Bloggs'


@app.route('/', methods=['GET', 'POST'])
def mainpage():

    name = default_name
    if request.method == 'POST':
        name = html.escape(request.form['name'], quote=True) ❶

    salted_name = salt + name
    name_hash = hashlib.sha256(salted_name.encode()).hexdigest()
    header = '<html><head><title>Identidock</title></head><body>'
    body = '''<form method="POST">
              Hello <input type="text" name="name" value="{0}">
              <input type="submit" value="submit">
              </form>
              <p>You look like a:
              <img src="/monster/{1}"/>
              '''.format(name, name_hash)
    footer = '</body></html>'

    return header + body + footer


@app.route('/monster/<name>')
def get_identicon(name):

    name = html.escape(name, quote=True) ❶
    image = cache.get(name)
    if image is None:
        print ("Cache miss", flush=True)
        r = requests.get('http://dnmonster:8080/monster/' + name + '?size=80')
        image = r.content
        cache.set(name, image)
```

```
        return Response(image, mimetype='image/png')

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

❶ 使用 html.escape 方法来净化用户输入。

现在，如果重新构建并测试我们的应用程序：

```
$ docker build -t identidock .
...
$ docker run identidock python tests.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.009s

OK
```

太棒了，问题已经解决。为了验证这一点，你可以通过使用新的容器来重新启动 identidock 程序（记得执行 docker-compose build 命令，从而确保 Compose 使用新的代码），并试图输入一些恶意数据。[2] 如果我们使用的不是简单的字符串组合，而是一个真正的模板引擎，那么它便会帮我们处理转义的事情，这个问题就不会出现了。

已经有一些测试了，现在可以增强我们的 cmd.sh 脚本，使它们能自动执行。把 cmd.sh 替换成以下内容：

```
#!/bin/bash
set -e

if [ "$ENV" = 'DEV' ]; then
  echo "Running Development Server"
  exec python "identidock.py"
elif [ "$ENV" = 'UNIT' ]; then
  echo "Running Unit Tests"
  exec python "tests.py"
else
  echo "Running Production Server"
  exec uwsgi --http 0.0.0.0:9090 --wsgi-file /app/identidock.py \
             --callable app --stats 0.0.0.0:9191
fi
```

现在就可以重新构建容器，以后只通过更改环境变量就能执行测试：

```
$ docker build -t identidock .
...
$ docker run -e ENV=UNIT identidock
Running Unit Tests
..
----------------------------------------------------------------------
Ran 2 tests in 0.010s

OK
```

注 2：很惭愧，我一直没有注意到这个问题，直到本书进入审阅阶段。我再一次明白了，即使看起来很简单的代码，测试也同样重要，并最好尽可能使用现成的且经过考验的代码和工具。

我们还可以写更多的单元测试。特别是 `get_identicon` 方法，目前还没有针对它的具体的测试。为了利用单元测试来验证这个方法，需要调用 dnmonster 和 Redis 服务的测试版本，或者通过使用*测试替身*（test double）。测试替身替代了真正提供服务的程序部分，通常只是一个*桩*（stub），只返回一个固定的答案（例如，一个股票价格服务的桩可以实现为总是返回“42”），或者是一个*模拟对象*（mock），它只能够以编写它的时候的预期来调用（例如，它只能够在某个特定的事务中被调用一次）。要了解更多测试替身的相关信息，请参阅 Python 模拟模块（https://docs.python.org/3/library/unittest.mock.html），以及专门的 HTTP 工具，如 Pact（https://github.com/realestate-com-au/pact）、Mountebnank（http://www.mbtest.org/）和 Mirage（https://mirage.readthedocs.org）。

把测试包含在镜像中

本章将 identidock 的测试放进了 identidock 镜像，这与 Docker 的哲学是一致的，它提倡从开发、测试到生产过程都只需要一个镜像。这也意味着，当镜像在不同的环境下运行时，我们都能轻松地对它进行测试，这对调试时排除问题非常有用。

这样做的缺点是创建出来的镜像会比较大，因为镜像需要包含测试代码和它所依赖的程序，例如测试框架的库。同时，这也扩大了*攻击面*；尽管看上去不太可能，但攻击者仍然可以利用测试工具或代码来入侵运行于生产环境的系统。

大多数情况下，相较于使用单一镜像所带来的简易性和可靠性，其稍大的镜像体积和理论上的安全风险不足以掩盖其优点。

下一步是把我们的测试放到持续集成服务器中自动运行，这样的话，当源码提交到源码控制服务器的时候，源码测试便会被自动执行，而这都是在源码进入准生产和生产环境前完成。

使用容器进行快速测试

所有的测试，特别是单元测试，运行速度必须快，才能鼓励开发者经常执行它们，否则等待测试结果势必阻碍开发者的工作。对于将会造成环境变化的测试，就可以使用容器技术，因为它能快速启动一个干净且被隔离的环境。举个例子，假设你有一组测试套件，需要使用一个已准备好测试数据的服务[3]。而每个使用该服务的测试都有可能以不同的方式改变数据，包括添加、删除或修改。编写这些测试的一种方法是，让每个测试在运行后尝试对数据进行清理，但这是有问题的；如果测试（或清理）失败，那么接下来的所有测试用到的数据都会被污染，使得排查出错原因变得复杂，而且要求对所测试的服务有所了解（那它就不再是黑盒子了）。另一种方法是，在每次测试之后彻底删除该服务，让每个测试都使用一个全新的服务。这种方法利用虚拟机来实现会非常慢，但容器却办得到。

注 3：像这样的测试很可能是系统或集成测试，而非单元测试。它们也可能是无法使用模拟对象模式的单元测试。很多单元测试专家会建议把诸如数据库的组件替换为模拟对象，但假如该组件稳定可靠，直接使用它们往往最简单，而且也是比较明智的做法。

容器适用于测试的另一个领域是在不同环境 / 配置下运行服务。如果你的软件在不同的 Linux 发行版以及不同的数据库上运行，只需为各个配置构建一个镜像，之后便可以在每个镜像上迅速执行你的测试了。这种做法需要注意的是，它并未考虑到各个发行版之间内核的差异。

8.2 创建Jenkins容器

Jenkins 是一个很流行的开源持续集成（CI）服务器。CI 服务器和相关的托管方案有很多，但在我们的 Web 应用项目中将采用 Jenkins，因为它非常流行。安装 Jenkins 后，每当有任何源码的修改被推送到 identidock 项目时，它便会检出那些修改、构建新镜像，并对镜像执行测试——包括单元测试及一些系统测试。然后，它会根据测试结果生成测试报告。

我们的方案将基于官方的 Jenkins 库中的镜像。我用过 `1.609.3` 版本，但 Jenkins 不断有新版本发布——放心尝试新版本，但我不能保证它在无需修改的情况下能照常工作。

为了能让我们的 Jenkins 容器构建镜像，我们将会把 Docker 套接字[4] 从主机挂载到容器内，实际上这是让 Jenkins 能够创建“同级”的容器。另一个方法是使用 Docker-in-Docker（DinD）技术，让 Docker 可以创建自己的“子”容器。图 8-1 展示了这两种方法的对比。

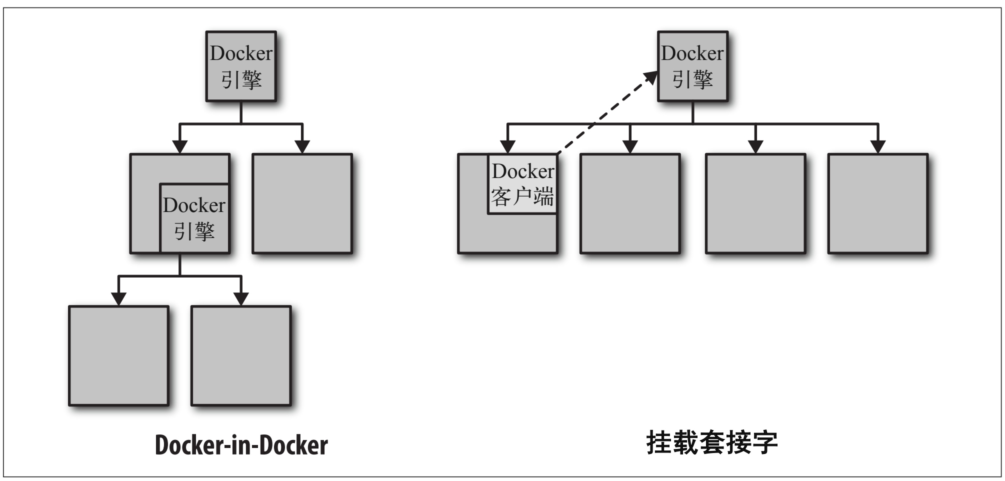


图 8-1：Docker-in-Docker 与挂载套接字

注 4：Docker 套接字是客户端和守护进程之间用来通信的端点。默认情况下，它是位于 /var/run/docker.sock 的 IPC 套接字，但 Docker 还支持基于网络地址的 TCP 套接字和 systemd 形式的套接字。本章假设使用位于 /var/run/docker.sock 的默认套接字。由于套接字是经由文件描述符访问的，只需把这个端点以数据卷形式挂载到容器内即可。

Docker-in-Docker

Docker-in-Docker（或 DinD）的意思是在 Docker 容器中运行 Docker 自己。要把它运行起来，需要做一些特殊的配置工作，主要是将容器运行于特权模式下，以及处理一些文件系统的相关事项。与其自己研究，不如使用 Jérôme Petazzoni 的 DinD 项目，这个项目可以在 https://github.com/jpetazzo/dind 找到，其中还详细描述了所需的工作步骤。使用 Jérôme 在 Docker Hub 上的 DinD 镜像，你就能快速入门：

```
$ docker run --rm --privileged -t -i -e LOG=file jpetazzo/dind
ln: failed to create symbolic link '/sys/fs/cgroup/systemd/name=systemd':
Operation not permitted
root@02306db64f6a:/# docker run busybox echo "Hello New World!"
Unable to find image 'busybox:latest' locally
Pulling repository busybox
d7057cb02084: Download complete
cfa753dfea5e: Download complete
Status: Downloaded newer image for busybox:latest
Hello New World!
```

DinD 与挂载套接字方式的主要区别在于，DinD 创建的容器与主机的容器是隔绝的；在 DinD 容器中执行 `docker ps` 命令只会显示 DinD Docker 服务创建的容器。与此相反，使用挂载套接字方式的话，执行 `docker ps` 命令将显示所有的容器，无论命令在哪里执行。

一般情况下，我比较喜欢挂载套接字的方式，因为它更简单，但在某些情况下，你可能需要 DinD 提供的更多隔离性。如果你选择使用 DinD，请注意以下几点。

- 你将有自己的缓存，起初构建镜像的时候会比较慢，而且你还不得不再次下载所有镜像。使用本地的寄存服务或复制镜像可以缓解这个问题。千万不要尝试挂载主机上的构建缓存；Docker 引擎认为只有它有权力访问缓存，因此两个 Docker 实例同时共享一个缓存将会引发严重后果。
- 因为容器必须在特权模式下运行，所以它不会比挂载套接字的方式更安全（如果攻击者能够提权，他就可以挂载任何设备，包括驱动器）。这个问题在未来将有所改善，因为 Docker 将会引入细粒度的权限控制，使用户可以选择 DinD 能够访问的设备。
- 由于 DinD 使用从 /var/lib/docker 目录挂载的数据卷，如果你在移除容器的时候忘记删除数据卷，那么你的磁盘空间很快就会被占满。

想要了解更多应谨慎使用 DinD 的原因，请参阅 jpetazzo 在 GitHub 上的文章（https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/）。

为了从主机挂载套接字，我们必须确保容器内的 Jenkins 用户具有足够的访问权限。首先创建一个名为 identijenk 的新目录，并在它下面用以下内容创建一个 Dockerfile：

```
FROM jenkins:1.609.3

USER root
RUN echo "deb http://apt.dockerproject.org/repo debian-jessie main" \
          > /etc/apt/sources.list.d/docker.list \
      && apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 \
```

```
            --recv-keys 58118E89F3A912897C070ADBF76221572C52609D \
        && apt-get update \
        && apt-get install -y apt-transport-https \
        && apt-get install -y sudo \
        && apt-get install -y docker-engine \
        && rm -rf /var/lib/apt/lists/*
RUN echo "jenkins ALL=NOPASSWD: ALL" >> /etc/sudoers

USER jenkins
```

这个 Dockerfile 基于 Jenkins 基础镜像安装 Docker 程序，并把无需密码的 `sudo` 权限给予 `jenkins` 用户。我们故意不把 `jenkins` 用户添加到 Docker 用户组，因此必须在所有 Docker 命令前加上 `sudo`。

不要使用 docker 用户组

要是不使用 `sudo`，可以把 `jenkins` 用户添加到主机的 `docker` 用户组。但问题是，我们需要找出并使用 CI 主机上 `docker` 用户组的 GID，并将其在 Dockerfile 上硬编码。这使得我们的 Dockerfile 不可移植，因为在不同主机上，`docker` 用户组的 GID 都不完全一样。为了避免它所带来的混乱和麻烦，`sudo` 是比较推荐的做法。

接下来构建镜像：

```
$ docker build -t identijenk .
...
Successfully built d0c716682562
```

测试一下：

```
$ docker run -v /var/run/docker.sock:/var/run/docker.sock \
        identijenk sudo docker ps
CONTAINER ID  IMAGE       COMMAND                 CREATED        STATUS  ...
a36b75062e06  identijenk  "/bin/tini -- /usr/lo"  1 seconds ago  Up Less tha...
```

在 `docker run` 命令中，我们挂载了 Docker 套接字，这样就能连接到主机上的 Docker 守护进程。使用旧版本的 Docker 时，容器中一般不会安装 Docker 程序，而会选择把它挂载进来。这样做的好处是可以使主机上和容器中的 Docker 版本保持一致。然而，从 1.7.1 版本开始，Docker 开始使用动态库，这意味着任何依赖关系都要一并挂载。与其花时间找出需要挂载及更新的程序库，不如简单地在镜像中安装 Docker。

现在，Docker 已经能够在容器中运行，接下来可以安装其他需要的东西，让这个 Jenkins 容器能真正起作用。将 Dockerfile 更新如下：

```
FROM jenkins:1.609.3

USER root
RUN echo "deb http://apt.dockerproject.org/repo debian-jessie main" \
            > /etc/apt/sources.list.d/docker.list \
        && apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 \
            --recv-keys 58118E89F3A912897C070ADBF76221572C52609D \
        && apt-get update \
```

```
    && apt-get install -y apt-transport-https \
    && apt-get install -y sudo \
    && apt-get install -y docker-engine \
    && rm -rf /var/lib/apt/lists/*
RUN echo "jenkins ALL=NOPASSWD: ALL" >> /etc/sudoers

RUN curl -L https://github.com/docker/compose/releases/download/1.4.1/\
docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-compose; \
    chmod +x /usr/local/bin/docker-compose ❶

USER jenkins
COPY plugins.txt /usr/share/jenkins/plugins.txt ❷
RUN /usr/local/bin/plugins.sh /usr/share/jenkins</plugins.txt
```

❶ 安装 Docker Compose，它将用于构建和运行我们的镜像。

❷ 把 plugins.txt 文件复制进来并执行一些处理工作，这个文件定义了安装到 Jenkins 的插件列表。

在 Dockerfile 同一目录下创建 plugins.txt 文件，内容包括：

```
scm-api:0.2
git-client:1.16.1
git:2.3.5
greenballs:1.14
```

前三个插件提供了接口，让我们能够访问 identidock 项目的 Git 仓库。“greenballs”插件将默认代表 Jenkins 构建成功的蓝色球替换成绿色的。

现在差不多已准备好启动我们的 Jenkins 容器，并开始进行镜像构建的配置，但首先应该创建一个数据容器，使我们的配置能持久保留：

```
$ docker build -t identijenk .
...
$ docker run --name jenkins-data identijenk echo "Jenkins Data Container"
Jenkins Data Container
```

我们刚才利用 Jenkins 镜像作为数据容器，因此可以肯定权限设置是正确的。当 `echo` 命令完成的时候，容器便会退出，但只要它还没被删除，就能用于 `--volumes-from` 参数。有关数据容器的详情，请参阅 4.5 节。

现在已准备好启动 Jenkins 容器：

```
$ docker run -d --name jenkins -p 8080:8080 \
    --volumes-from jenkins-data \
    -v /var/run/docker.sock:/var/run/docker.sock \
    identijenk
75c4b300ade6a62394a328153b918c1dd58c5f6b9ac0288d46e02d5c593929dc
```

如果在浏览器中打开 http://localhost:8080，应该会看到 Jenkins 正在初始化。稍后会开始配置 identidock 项目的镜像构建及测试。但在此之前，我们需要对 identidock 项目做一个小改动。目前，项目中的 docker-compose.yml 文件启动的是开发版的 identidock，但我们即将开发一些系统测试，并且希望这些测试能够在更接近生产环境的环境中运行。出于这个原因，我们需要创建一个新的 jenkins.yml 文件，我们将会用它在 Jenkins 中启动 identidock

的生产版本：

```
identidock:
  build: .
  expose:
   - "9090"  ❶
  environment:
    ENV: PROD  ❷
  links:
   - dnmonster
   - redis

dnmonster:
  image: amouat/dnmonster:1.0

redis:
  image: redis:3.0
```

❶ 由于 Jenkins 位于同一级容器，我们可以连接它，却无需在主机上发布端口。其中的 expose 指令主要是为了记录而已；即使没有它，如果你没有修改过默认的网络设置，也仍然可以从 Jenkins 访问 identidock 容器。
❷ 把环境设置为生产环境。

这个文件需要添加到 identidock 源码库，Jenkins 将会从该库中获取源码。假如你之前已经配置了一个自己的源码库，你可以把它添加到那里，或者使用现有的源码库（https://github.com/using-docker/identidock）。

现在已准备就绪，可以开始配置 Jenkins 的构建工作了。打开 Jenkins 的网页界面（http://localhost:8080），并按照以下的说明操作。

(1) 点击“create new jobs”链接。
(2) 输入“identidock”作为“Item name”，选择“Freestyle project”，然后点击 OK。
(3) 配置“Source Code Management”选项。如果你使用的是公开的 GitHub 仓库，只需选择“Git”并输入仓库的 URL。如果你使用的是私有仓库，你将需要配置某种形式的身份认证 [包括 BitBucket 在内的一些仓库，均备有*部署钥匙*（deployment keys）可用于设置只读访问，可以满足这个需要]。除此以外，也可以使用 GitHub 上的版本（https://github.com/using-docker/identidock）。
(4) 点击“Add build step”，然后选择“Execute shell”。在“Command”框中，输入以下内容：

```
#Default compose args
COMPOSE_ARGS=" -f jenkins.yml -p jenkins "

#Make sure old containers are gone
sudo docker-compose $COMPOSE_ARGS stop  ❶
sudo docker-compose $COMPOSE_ARGS rm --force -v

#build the system
sudo docker-compose $COMPOSE_ARGS build --no-cache
sudo docker-compose $COMPOSE_ARGS up -d

#Run unit tests
sudo docker-compose $COMPOSE_ARGS run --no-deps --rm -e ENV=UNIT identidock
ERR=$?
```

```
#Run system test if unit tests passed
if [ $ERR -eq 0 ]; then
  IP=$(sudo docker inspect -f {{.NetworkSettings.IPAddress}} \
          jenkins_identidock_1)  ❷
  CODE=$(curl -sL -w "%{http_code}" $IP:9090/monster/bla -o /dev/null) || true  ❸
  if [ $CODE -ne 200 ]; then
    echo "Site returned " $CODE
    ERR=1
  fi
fi

#Pull down the system
sudo docker-compose $COMPOSE_ARGS stop
sudo docker-compose $COMPOSE_ARGS rm --force -v

return $ERR
```

❶ 注意这里使用了 `sudo` 来调用 Docker Compose，因为之前说过，Jenkins 用户不在 `docker` 用户组里。

❷ 通过 `docker inspect` 命令来找出 identidock 容器的 IP 地址。

❸ 利用 `curl` 访问 identidock 服务，并查到它返回的 HTTP 码是 200，这代表它能正常工作。请注意，我们使用路径 `/monster/bla` 以确保 identidock 可以连接到 dnmonster 服务。

你也可以在 GitHub 上找到这段代码（https://github.com/using-docker/ci-testing）。像这样的脚本一般会与其他代码一起提交到源码控制系统，但就我们这个例子而言，简单地把它粘贴到 Jenkins 就可以了。

现在，点击“Save”后再点击“Build Now”，就能开始测试了。点击 build ID，然后选择“Console Output”，就能查看构建时的详细信息。你应该会看到类似图 8-2 的画面。



图 8-2：Jenkins 构建成功

目前为止，Jenkins 的表现都相当不错。我们成功地将 Docker 运行起来，并针对我们的应用让它执行了单元测试，以及一个简单的“冒烟测试”。如果它是一个真正的应用，我们会继续开发一套完整的测试，以确保应用程序正常运作，并能处理各种不同的输入。但由于它只是一个简单的演示，做到这种程度就已经足够了。

触发构建

直到现在，构建都是由手动点击“Build Now”所触发的。如果当 GitHub 项目有新的源码提交时，构建便会自动发生，那么将是一个很显著的提升。要实现这个功能，可以启用位于 identidock 配置中的“Poll SCM”方法，并在文本框中输入“H/5 * * * *”。这样做之后，Jenkins 便会每 5 分钟检查一下源码库有没有任何变化，如果发现变化，就会安排执行一次构建。

这是一个简单易行的解决方法，但它有点浪费资源，而且构建会有最多 5 分钟的持续滞后。更好的方法是配置仓库，让它能够在变化发生的时候通知 Jenkins。这个功能可以通过 BitBucket 或 GitHub 的 Web Hook 实现，但要求 Jenkins 的服务器能够在公共网络访问得到。

使用 Docker Hub 镜像

说到这里，有些人可能会问：“为什么我们要构建镜像呢？”如果你跟着上一节做了，你应该已经有一个在 Docker Hub 上的自动构建，当源码提交到仓库时，自动构建就会被触发。你可以利用这个特性，并通过 Docker Hub 的 Webhook 功能，让 Docker Hub 的自动构建结束后自动启动 Jenkins 的构建。这样做的话，我们的脚本就只需下载而不需构建镜像。但这也需要 Jenkins 服务器能够在公共网络被访问得到。

这个方案或许对只需创建独立 Docker 镜像的小型项目有用，但对于大型项目，它们可能更需要自己能够控制构建，以获得更快的速度及更高的安全性。

8.3 推送镜像

既然 identidock 镜像已经通过测试了，现在是时候找方法让它进入工作流的下一环节。首先给镜像一个标签，并将它推送到寄存服务。完成这一步之后，工作流的下一环节就会有一个可用的镜像，可以把它推送到准生产或生产环境中。

8.3.1 给镜像正确的标签

在一个基于容器的工作流中，给镜像正确的标签对维护其控制权和出处至关重要。一旦弄错了，你就会发现，在生产环境中有些镜像很难回溯它的构建源头（虽然并非不可能，但或许会非常困难），这就造成了调试和维护的不必要的麻烦。对于任何一个镜像，我们应当能够确切指出用于创建它的 Dockerfile 和构建上下文。[5]

注 5：请注意，这并不保证你能够重新创建一个相同的容器，因为依赖关系可能已经发生改变。在 13.6.3 节中你会看到怎样应对这种情况。

任何时候标签都可以被覆盖和改变。因此，必须由你来建立和实施一个可靠的流程来规定镜像的标签和版本准则。

在我们的范例程序中将对镜像添加两个标签：源码库的 git 散列值以及 newest。在这个方法中，newest 标签会一直用于已通过测试的最新版本，而我们可以使用 git 散列值恢复任何镜像的构建文件。鉴于之前在 7.1 节的警示信息“当心 latest 标签”中讨论过的问题，我刻意避免了使用 latest 标签。现在让我们更新 Jenkins 的构建脚本：

```
#Default compose args
COMPOSE_ARGS=" -f jenkins.yml -p jenkins "

#Make sure old containers are gone
sudo docker-compose $COMPOSE_ARGS stop
sudo docker-compose $COMPOSE_ARGS rm --force -v

#build the system
sudo docker-compose $COMPOSE_ARGS build --no-cache
sudo docker-compose $COMPOSE_ARGS up -d

#Run unit tests
sudo docker-compose $COMPOSE_ARGS run --no-deps --rm -e ENV=UNIT identidock
ERR=$?

#Run system test if unit tests passed
if [ $ERR -eq 0 ]; then
  IP=$(sudo docker inspect -f {{.NetworkSettings.IPAddress}} \
          jenkins_identidock_1)
  CODE=$(curl -sL -w "%{http_code}" $IP:9090/monster/bla -o /dev/null) || true
  if [ $CODE -eq 200 ]; then
    echo "Test passed - Tagging"
    HASH=$(git rev-parse --short HEAD) ❶
    sudo docker tag -f jenkins_identidock amouat/identidock:$HASH ❷
    sudo docker tag -f jenkins_identidock amouat/identidock:newest ❷
    echo "Pushing"
    sudo docker login -e joe@bloggs.com -u jbloggs -p jbloggs123 ❸
    sudo docker push amouat/identidock:$HASH ❹
    sudo docker push amouat/identidock:newest ❹
  else
    echo "Site returned " $CODE
    ERR=1
  fi
fi

#Pull down the system
sudo docker-compose $COMPOSE_ARGS stop
sudo docker-compose $COMPOSE_ARGS rm --force -v

return $ERR
```

❶ 取得 git 散列值的短式。

❷ 添加标签。

❸ 登录寄存服务。

❹ 把镜像推送到寄存服务。

注意，你需要针对将要推送的目标仓库来命名标签。假设你的仓库运行在 `myhost:5000` 之上，那么标签应为 `myhost:5000/identidock:newest`。同样，你还需要修改 `docker login` 命令中的用户信息。

如果你现在重新进行构建，会发现脚本已经懂得添加标签并把镜像推送到寄存服务器，准备好进入工作流的下一环节。对于我们的示例应用来说，这样已经很不错了；对大多数希望建立这套流程的项目而言，这也是个很好的范例。但随着程序的复杂度增加，可能你需要使用更多标签及更具描述性的名称。`git describe` 是个很有用的命令，它能够基于标签生成更有意义的名称。

找出镜像的所有标签

镜像的每个标签都是独立存储的。这意味着，要找出镜像的所有标签，就需要过滤基于镜像 ID 的完整镜像列表。以标签为 `amouat/identidock:latest` 的镜像作为例子，要找出它的所有标签，可以这样做：

```
$ docker images --no-trunc | grep \
    $(docker inspect -f {{.Id}} amouat/identidock:newest)
amouat/identidock   51f6152  96c7b4c094c8f76ca82b6206f...
amouat/identidock   newest   96c7b4c094c8f76ca82b6206f...
jenkins_identidock  latest   96c7b4c094c8f76ca82b6206f...
```

可以看到，同一镜像也有另一个标签，叫作 `51f6152`。

请记住，你只会看到存在于你的镜像缓存中的标签。举个例子，假如下载标签为 `debian:latest` 的镜像，那么是不会有 `debian:7` 标签的，即使（在写作的时候）它们的镜像 ID 一样。同样，如果我同时有 `debian:latest` 和 `debian:7` 两个镜像，并下载了新版本 `debian:latest`，那么已标签为 `debian:7` 的镜像将不会受到影响，它将仍然与之前的镜像关联。

8.3.2 准生产及生产环境

一旦图像通过测试，附上标签并推送到寄存服务器，它就需要供工作流的下一阶段使用，可能是准生产或生产环境。我们有多个实现方式，包括使用寄存服务的 webhook 通知（https://docs.docker.com/registry/notifications/），或利用 Jenkins 调用下一步。

8.3.3 镜像数量激增的问题

在生产系统中，你需要解决镜像数量激增的问题。你需要定期清理 Jenkins 服务器的镜像，还需要控制寄存服务器中的镜像数目，否则服务器便会迅速被陈旧过时的镜像占满。其中一个解决方法是，设定一个时间，将该时间之前的镜像全部删除，如果空间允许的话，也可以把它们备份起来。[6] 还有其他产品能够提供更先进的功能，像 CoreOS Enterprise

注 6：在撰写这部分的时候，如果 Docker 寄存服务器是在本地搭建的话，鉴于当时删除功能尚未实现，这一点说起来容易做起来难。在实现之前还有一些难点需要克服，分发路线图（https://github.com/docker/distribution/blob/master/ROADMAP.md）中对这些问题有详细描述。

Registry 或 Docker Trusted Registry，它们都具备一些管理仓库的高级功能。

> **确保测试的镜像是正确的**
>
> 你必须确保用于测试的容器镜像与运行在生产环境中的是同一个，这是非常重要的。千万不要在测试环境中用 Dockerfile 构建镜像后，又在生产环境中重新构建一次——一定要确保真正运行时的镜像是之前测试通过的，并且其中没有任何无意间的改动。出于这个原因，应用某种形式的寄存服务或存储，使镜像可以运行在测试、准生产和生产环境里，是非常必要的。

8.3.4 使用Docker部署Jenkins slaves

随着构建的需求增加，运行测试所需的资源也越来越多。Jenkins 有一个概念叫作“build slaves”，它主要是一个执行任务的集群，让 Jenkins 将构建任务分派到集群里的机器。

如果你希望使用 Docker 来动态部署 slaves，请参阅 Docker 的 Jenkins 插件（https://wiki.jenkins-ci.org/display/JENKINS/Docker+Plugin）。

8.4 备份Jenkins数据

由于我们的 Jenkins 服务使用了数据容器，备份 Jenkins 的数据非常简单：

```
$ docker run --volumes-from jenkins-data -v $(pwd):/backup \
         debian tar -zcvf /backup/jenkins-data.tar.gz /var/jenkins_home
```

这个命令会在 $(pwd)/backup 目录下产生一个 jenkins-data.tar.gz 文件。在执行这个命令之前，你可能需要先把 Jenkins 容器停止或暂停。之后你就可以执行如下的命令来创建一个新的数据容器，并在它内部把备份数据解开：

```
$ docker run --name jenkins-data2 identijenk echo "New Jenkins Data Container"
$ docker run --volumes-from jenkins-data2 -v $(pwd):/backup \
           debian tar -xzvf /backup/backup.tar
```

但是这个方法需要你知道容器的详细挂载目录，而这是可以通过检视容器来自动处理的，因此也可以使用诸如 docker-backup（https://github.com/discordianfish/docker-backup）的工具来帮助你完成这个任务，而我期待未来 Docker 会对这种工作流程有更多的支持。

8.5 持续集成的托管解决方案

持续集成的托管解决方案有不少，从云端提供与维护 Jenkins 服务的公司，到更专业的解决方案提供商，如 Travis（https://travis-ci.org）、Wercker（http://www.wercker.com/）、CircleCI（https://circleci.com/）以及 drone.io（https://drone.io/）。大多数解决方案似乎都是针对在某种特定的语言栈上运行单元测试，而不是面向容器进行系统测试。目前这方面好像有一些进展，我期待能看到测试 Docker 容器的产品早日面世。

8.6 测试与微服务

如果你使用 Docker，那么你很可能已经采用了微服务架构。当测试一个微服务架构时，测试可以分为不同的级别，如何测试以及测试什么将由你来决定。基本的测试框架可以包括以下内容。

单元测试

每个服务[7] 都应该有一套详尽的单元测试。单元测试应该只测试一小块独立的功能。你可以使用测试替身来代替该功能所依赖的其他服务。由于测试的数量很多，为了鼓励开发者经常执行测试，以及避免把时间花在等待测试返回结果上，尽量减少测试的运行时间非常重要。在你的系统的所有测试当中，单元测试所占的比例应该最高。

组件测试

这一类测试的级别，可以是针对各个服务的外部接口，或是针对一组服务的子系统。在这两种情况下，你都很可能会发现测试中存在依赖于其他服务的部分，这时候便需要按照前文所述，将其替换为测试替身。你或许还会发现，通过这些服务的 API 提供的指标和日志数据有助于测试，但你必须确保它们使用的命名空间与正式执行时的 API 不是同一个（例如使用不同的 URL 前缀）。

端到端测试（end-to-end tests）

这种测试确保整个系统正常运作。由于执行这种测试的成本相当昂贵（资源和时间方面），它们应该为数不多——你绝对不想花几个小时来执行它们，从而导致部署和发布问题修正的时间被延误（可以考虑我将介绍的*定时任务*）。也有可能部分系统不可测，或者测试的成本极昂贵，这种情况下可能仍然需要使用测试替身（你不会在测试中真正发射核导弹吧）。我们在 identidock 中的测试便属于一种端到端测试；这个测试把整个系统从头到尾运行一遍，而且没有使用测试替身。

此外，你可能还需要考虑下列测试。

使用方契约测试（consumer-contract tests）

这类测试也称为使用方驱动契约，由服务的*使用方*负责编写，它定义了对于该服务预期的输入和输出数据，还可以包括可能出现的副作用（状态的改变）和性能的预期。每个该服务的使用方应该有单独的契约。这种测试的好处是，它让开发该服务的人员知道何时出现有可能对兼容性有影响的风险；当契约测试失败时，他们会选择或者对服务进行修改，或者需要与服务使用方的开发人员一起商讨怎么改变契约。

集成测试

这类测试的目的是确保每个组件之间的通信渠道均运行正确。这类测试对微服务架构非常重要，因为其中组件与组件之间需要大量的紧密连接与协调，要比传统的单一架构系统高出一个数量级。但是，你可能会发现大部分的通信渠道都已经由组件测试和端到端测试覆盖了。

注 7：通常情况下，每个服务都有一个容器，如果服务需要更多资源，也可能使用多个容器。

定时任务

> 由于需要尽量保持持续集成能在短时间内执行完毕，我们往往没有足够的时间来运行全方位的测试，例如针对不常见的配置或针对不同平台。不过，我们可以安排在晚上执行它们，那时候能有剩余资源可供使用。

这些测试中的许多可以被分类为*寄存服务前*（preregistry）和*寄存服务后*（postregistry），取决于测试是在镜像添加到寄存服务之前还是之后。例如，单元测试属于寄存服务前：假如单元测试失败，那么镜像就不应该被推送到寄存服务中。这一情况同样适用于一些使用方契约测试和一些组件测试。另一方面，在进行端到端测试前，镜像必须已被推送到寄存服务。如果寄存服务后的测试失败，那么就要认真思考下一步该怎么做了。虽然遇到这种情况时，任何新的镜像都不应该再被推送到生产环境（或者如果它们已经部署了，就应该回滚），但实际上问题可能是由其他镜像造成的，或者是由旧的镜像，甚至有可能是由新镜像之间的相互影响造成的。这类问题可能需要更深层的调查和思考才能得到正确解决。

在生产环境中测试

最后，你可能想了解一下如何在生产环境中进行测试。别担心，这并不像听起来那么疯狂。尤其是在生产环境中，你将要处理大量用户以及各种不同的环境和配置，测试它们并不容易，但正因为如此才更有测试的必要。

一种常见的测试方法叫作蓝 / 绿部署（blue/green deployment）。假设我们打算更新目前在生产环境中的一个服务，暂且把这个版本称为“蓝色”版本，而更新后的版本称之为“绿色”版本。虽然可以简单地用绿色版本替换蓝色版本，但我们不这样做，而是选择在一段时间内同时运行这两个版本。当绿色版本运行起来后，我们就把所有网络通信一下转到绿色版本去。然后，我们监测系统，观察它的行为是否有任何意外变化，例如错误率或网络延迟的增加。如果不满意新版本的表现，我们所要做的只是把生产版本立即转回蓝色版本。一旦我们对系统运行顺畅感到满意，就可以将蓝色版本关闭。

其他的方法也遵循类似的原理，即同时运行新旧两个版本。在 *A/B 测试*，或*多变量测试*（multivariate testing）中，两个（或多个）版本的服务在测试期间同时运行，而使用这个服务的用户会被随机分配到其中一个。测试期间会收集某些统计数据，并根据测试结束时的结果保留其中一个版本。在*渐增式部署*（ramped deployment）中，服务的新版本只会提供给一小部分用户。如果这些用户没有发现问题，那么这个新版本就会逐步向越来越多的用户提供。至于*阴影测试*（shadowing），所有请求都会让两个版本的服务进行处理，但真正使用的只有来自旧版本即稳定版的结果。通过比较新旧两个版本的结果，就能确保新版本的行为与旧版本一模一样（或者有所差异，不过是符合预期的）。阴影测试尤其适用于测试没有改变功能的新版本，例如只是性能提升的新版本。

8.7 总结

本章的核心思想就是容器天生适用于持续集成与持续交付的工作流。当应用于这些场景时，有几件事情必须谨记，其中最重要的就是在工作流中的每一步都必须使用同一个镜

像，而不应该重新构建。尽管如此，使容器能够适配现有的持续集成工具应该不会有太多问题，而且未来将会有更专业的工具出现在这个领域。

如果你正尝试使用大型的微服务架构，不妨花更多的时间来思考测试方法，以及研究本章中提到的一些技术。

第 9 章

部署容器

现在是时候涉足业务方面的事情了，我们要开始思考如何真正在生产环境中使用 Docker。在撰写这本书的时候，大家都在谈论 Docker，很多人在试验它，但真正把 Docker 应用于生产的相对来说还是少数。尽管批评者有时会指出这是 Docker 的失败，但他们似乎忽略了几个关键点。虽然 Docker 的出现时间相对较晚，但已经有很多人开始在生产环境中使用它（包括 Spotify、Yelp 和百度），这是非常令人鼓舞的事情，而即使只是在开发和测试中使用它的人也获益良多。

话虽如此，今天在生产环境中使用容器是完全可能且合理的。较大型的项目和机构可能希望从小入手，慢慢把规模扩大，但对于大部分项目而言，它已经是个可行和直接的解决方案了。

就目前的情况来看，部署容器的最常用方法是先把虚拟机部署好，然后在虚拟机上启动容器。这并不是理想的解决方案——这样做会产生大量开销，拖延扩展，并且强制用户进行多容器粒度的部署。在虚拟机上运行容器的主要原因只是为了安全。你必须保证客户无法访问其他客户的数据或网络通信，然而容器目前在隔离性方面提供的保障还比较脆弱。此外，如果有某个容器独占了内核资源，或者造成内核错误，这将影响在同一台主机上运行的所有容器。即使大多数专门的解决方案，如谷歌的 Google Container Engine（GKE）和亚马逊的 Amazon EC2 Container Service（ECS），内部也仍然使用虚拟机。不过目前有两个例外，一个是 Giant Swarm，另一个是 Joyent 公司的 Triton，后面将会讨论到。

本章将介绍如何把我们简单的 Web 应用部署到一系列的云平台，以及专门的 Docker 托管服务上。本章还会涉及在公有云和私有网络的生产环境中运行容器时将会遇到的一些问题和采用的技术手段。

本章的代码见于本书的 GitHub 仓库（https://github.com/using-docker/deploying-containers）。我们不会再用之前的 Python 代码构建镜像，但还会继续使用已经创建好的镜像。你可以选择使用你自己的 identidock 镜像，或者直接使用 `amouat/identidock` 仓库。

你可以通过 `v0` 标签获取这一章开始时的代码：

```
$ git clone -b v0 \
  https://github.com/using-docker/deploying-containers/
...
```

后面的标签代表代码在本章中演变的各个阶段。

除此以外，你也可以在 GitHub 项目的 Releases 页面（https://github.com/using-docker/deploying-containers/releases）下载各个标签的代码。

9.1　通过Docker Machine配置资源

Docker Machine 是一种配置新资源的最快且最简单的方式，并能让容器在其上运行。Docker Machine 能够创建服务器、在服务器上安装 Docker，以及配置本地 Docker 客户端，让它们能够访问服务器。Docker Machine 自带了很多驱动，能够适配大部分主流的云服务提供商（包括 AWS、谷歌的 Google Compute Enginer、微软 Azure、Digital Ocean）以及 VMWare 和 VirtualBox。

注意 beta 软件

撰写这部分的时候，Docker Machine 尚处于 beta 阶段（我测试的版本是 0.4.1）。这意味着你很可能会遇到错误和缺少功能，但它应该仍然可以使用，而且还比较稳定。不过，这也意味着你看到的命令和语法相比于这里可能会略有改变。出于这个原因，我不建议在生产环境中正式使用 Docker Machine，话虽如此，它在测试和实验中还是非常有帮助的。

（是的，这个警告几乎对本书的所有部分都适用，我只是觉得是时候再次把它指出而已……）

现在来看如何使用 Docker Machine 将 identidock 在云端运行起来。首先，你需要在本地机器上安装 Docker Machine。如果你是通过 Docker 工具箱安装 Docker 的话，那就已经有 Docker Machine 了。如果不是，还可以从 GitHub 下载二进制档（https://github.com/docker/machine/releases），然后把它放在执行路径中（例如 /usr/local/bin/docker-machine）。完成这些步骤后，就可以开始执行命令了：

```
$ docker-machine ls
NAME       ACTIVE   DRIVER       STATE     URL                         SWARM
default             virtualbox   Running   tcp://192.168.99.100:2376
```

执行上述代码后不一定会得到任何输出，这依 Docker Machine 侦测到的主机而定。就本例而言，它找到了本地的 `boot2docker` 虚拟机。下一步要做的是在云端添加一台主机。我

将以 Digital Ocean 作为示范，不过 AWS 和其他云服务提供商也是大同小异。你需要在网上注册并生成个人的访问令牌访问（“Applications & API”页面，https://cloud.digitalocean.com/settings/applications）才能继续。它将会对你所使用的资源收取费用，因此当你使用完毕时，一定要把机器删除：

```
$ docker-machine create --driver digitalocean \
    --digitalocean-access-token 4820... \
    identihost-do
Creating SSH key...
Creating Digital Ocean droplet...
To see how to connect Docker to this machine, run: docker-machine env identi...
```

现在我们已经在 Digital Ocean 上创建了一个 Docker 主机。接下来要做的就是按照输出中给出的命令，让本地的客户端指向它：

```
$ docker-machine env identihost-do
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://104.236.32.178:2376"
export DOCKER_CERT_PATH="/Users/amouat/.docker/machine/machines/identihost-do"
export DOCKER_MACHINE_NAME="identihost-do"
# Run this command to configure your shell:
# eval "$(docker-machine env identihost-do)"
$ eval "$(docker-machine env identihost-do)"
$ docker info
Containers: 0
Images: 0
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 0
 Dirperm1 Supported: false
Execution Driver: native-0.2
Logging Driver: json-file
Kernel Version: 3.13.0-57-generic
Operating System: Ubuntu 14.04.3 LTS
CPUs: 1
Total Memory: 490 MiB
Name: identihost-do
ID: PLDY:REFM:PU5B:PRJK:L4QD:TRKG:RWL6:5T6W:AVA3:2FXF:ESRC:6DCT
Username: amouat
Registry: https://index.docker.io/v1/
WARNING: No swap limit support
Labels:
 provider=digitalocean
```

可以看到，我们已连接到运行在 Digital Ocean 上的一台 Ubuntu 主机。如果现在执行 `docker run hello-world`，它就会在云端的服务器上执行。

要运行 identidock 应用，可以使用第 6 章完结前的 docker-compose.yml，或者使用以下的 docker-compose.yml，这个文件使用的镜像是 Docker Hub 上的 identidock：

```
identidock:
  image: amouat/identidock:1.0
```

```
    ports:
     - "5000:5000"
     - "9000:9000"
    environment:
      ENV: DEV
    links:
      - dnmonster
      - redis
  dnmonster:
    image: amouat/dnmonster:1.0
  redis:
    image: redis:3
```

注意，如果 Compose 文件包含 `build` 指令，镜像构建便会在云端的服务器上发生。任何数据卷挂载的指令都必须移除，因为它们指向的将会是云端服务器的磁盘，而不是本地计算机上的。

跟往常一样执行 Compose：

```
$ docker-compose up -d ❶
...
Creating identidock_identidock_1...
$ curl $(docker-machine ip identihost-do):5000 ❷
<html><head><title>Hello...
```

❶ 这将需要一段时间，因为它需要先下载并构建所需的镜像。
❷ 可以利用 `docker-machine ip` 命令来查找我们的 Docker 主机运行的位置。

所以，现在 identidock 已经在云端运行起来了，任何人都能够访问。[1] 这太棒了，我们这么快就能够把它运行起来，但还有一些问题有待修正。我们注意到，应用程序正在使用开发用的 Python Web 服务器，端口为 5000。我们应该改用生产环境的服务器，但如果在应用前把反向代理或负载均衡用上就更好了，这样便能够对 identidock 的架构进行更改，而无需改变对外的 IP 地址。Nginx 支持负载均衡，因此通过 nginx 就很容易启动多个 identidock 实例，然后将流量平均分配到各实例。

对 identidock 进行冒烟测试

本书中通过 curl 命令来确保 identidock 服务正常运作。然而，仅仅获取首页并不是个很好的测试，这样做只能证明 identidock 容器已经运行起来而已。更好的测试应该是获取一个 identicon 图像，这样就能证明 identidock 和 dnmonster 两个容器都在工作中并能互相沟通。你可以用以下方法达成：

```
$ curl localhost:5000/monster/gordon | head -c 4
�PNG
```

我们使用了 Unix 的 `head` 命令来获取图像的前四个字符，从而避免把二进制数据打印到终端上。

注 1：一些服务供应商需要你先在防火墙把 5000 端口打开，如 AWS。

9.2 使用代理

让我们利用 nginx 在 identidock 服务的前面创建一个反向代理。现在创建一个名为 identiproxy 的新文件夹，并创建以下的 Dockerfile：

```
FROM nginx:1.7

COPY default.conf /etc/nginx/conf.d/default.conf
```

同时用以下内容创建一个 default.conf 文件：

```
server {
    listen       80;
    server_name  45.55.251.164; ❶

    location / {

      proxy_pass  http://identidock:9090; ❷
      proxy_next_upstream error timeout invalid_header http_500 http_502
                          http_503 http_504;
      proxy_redirect off;
      proxy_buffering off;
      proxy_set_header        Host            45.55.251.164; ❶
      proxy_set_header        X-Real-IP       $remote_addr;
      proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

❶ 把这里的 IP 地址替换成你的 Docker 主机的 IP 地址或它的域名。
❷ 重定向所有网络流量到 identidock 容器。我们将使用连接来实现它。

如果你的 Docker Machine 还在运行并连接至云端的服务器，现在就可以在云端的服务器构建镜像：

```
$ docker build --no-cache -t identiproxy:0.1 .
Sending build context to Docker daemon 3.072 kB
Sending build context to Docker daemon
Step 0 : FROM nginx:1.7
 ---> 637d3b2f5fb5
Step 1 : COPY default.conf /etc/nginx/conf.d/default.conf
 ---> 2e82d9a1f506
Removing intermediate container 5383f47e3d1e
Successfully built 2e82d9a1f506
```

我们很容易就会忘记正在连接到一个远程的 Docker 引擎，但现在镜像已经在远程的服务器上了，而不是在你的本地开发机器上。

现在，我们可以返回到 `identidock` 文件夹，并创建一个新的 Compose 配置文件来测试它。用以下内容创建一个 prod.yml 文件：

```
proxy:
  image: identiproxy:0.1  ❶
  links:
```

```
      - identidock
    ports:
     - "80:80"
  identidock:
    image: amouat/identidock:1.0
    links:
     - dnmonster
     - redis
    environment:
      ENV: PROD ❷
  dnmonster:
    image:amouat/dnmonster:1.0 ❶
  redis:
    image: redis:3 ❶
```

❶ 请注意，我对所有镜像都使用了标签。在生产环境中，你要注意你运行的容器是什么版本。使用 `latest` 标签的做法很不妥，因为这样你将很难或甚至无法分辨容器中运行的应用是哪个版本。

❷ 注意，我们不再需要声明 identidock 容器将要打开哪些端口（代理容器才需要这样做），因此把环境变量改成启动生产环境的 Web 服务器。

在 Compose 中使用 `extends` 指令

如果 YAML 文件变得冗长，可以使用 `extends` 关键字在不同的环境之间共享配置信息。例如，我们可以定义一个文件叫作 common.yml，它包含以下内容：

```
identidock:
  image: amouat/identidock:1.0
  environment:
    ENV: DEV
dnmonster:
  image: amouat/dnmonster:1.0
redis:
  image: redis:3
```

然后可以把 prod.yml 重写如下：

```
proxy:
  image: identiproxy:0.1
  links:
    - identidock
  ports:
   - "80:80"
identidock:
  extends:
    file: common.yml
    service: identidock
  environment:
    ENV: PROD
dnmonster:
  extends:
    file: common.yml
```

```
        service: dnmonster
    redis:
      extends:
        file: common.yml
        service: redis
```

extends 关键字会从公用的文件中把相应的配置拉进来。prod.yml 中的配置将覆盖 common.yml 中的配置。links 和 volumes-from 的值是**不会**被继承的，以免意外的状况发生。出于这个原因，在我们的示例中，使用 extends 实际上会导致 prod.yml 文件更冗长，虽然自动继承基础文件改动的这个优点还是存在的。本书中避免使用 extends 的主要原因只不过是希望保持范例独立。

停止旧版本并启动新版本：

```
$ docker-compose stop
Stopping identidock_identidock_1... done
Stopping identidock_redis_1... done
Stopping identidock_dnmonster_1... done
Starting identidock_dnmonster_1...
Starting identidock_redis_1...
Recreating identidock_identidock_1...
Creating identidock_proxy_1...
```

现在来测试一下；它现在应该能够在默认的 80 端口响应，而不是 9090 端口：

```
$ curl $(docker-machine ip identihost-do)
<html><head><title>Hello...
```

非常好！现在我们的容器已经位于代理的后面了，这使得我们可以做很多事情，诸如对一组 identidock 实例实行负载均衡，或将 identidock 迁移到新的主机而不需要更改访问的 IP 地址（只要代理仍旧在原来的主机上，并把它的配置更新一下）。此外，安全性也得到增强，因为应用程序容器只可以通过代理来访问，并且再没有任何端口暴露于互联网上。

不过我们还可以做得更好。主机的 IP 地址和容器名称被固定在代理的镜像当中，这是很烦人的；如果我们打算不再使用“identidock”这个名称而改用别的，或者希望把 identiproxy 给别的服务使用，那么便需要重新构建镜像，或者通过数据卷的方式把配置重写。我们希望让这些参数通过环境变量来配置。我们无法直接在 nginx 中使用环境变量，但可以写一个脚本，通过它来动态生成配置文件，然后启动 nginx。我们需要回到 identiproxy 文件夹，把其中的 default.conf 里已硬编码的变量换成占位符：

```
server {
    listen       80;
    server_name {{NGINX_HOST}};

    location / {

      proxy_pass {{NGINX_PROXY}};
      proxy_next_upstream error timeout invalid_header http_500 http_502
                          http_503 http_504;
      proxy_redirect off;
```

```
            proxy_buffering off;
            proxy_set_header        Host            {{NGINX_HOST}};
            proxy_set_header        X-Real-IP       $remote_addr;
            proxy_set_header        X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }
```

我们还需要创建以下的 entrypoint.sh 文件，它将执行变量的更换：

```
#!/bin/bash
set -e

sed -i "s|{{NGINX_HOST}}|$NGINX_HOST|;s|{{NGINX_PROXY}}|$NGINX_PROXY|" \
  /etc/nginx/conf.d/default.conf ❶
cat /etc/nginx/conf.d/default.conf ❷
exec "$@" ❸
```

❶ 使用 sed 命令来执行更换的动作。虽然这样做不太漂亮，但对于达到我们的目的已经足够了。注意我们使用了 | 符号而不是 /，这样做是为了避免与 URL 中的斜杠混淆。
❷ 把已替换完成的模板打印到日志中，为了方便调试。
❸ 执行任何传入的 CMD 命令。Nginx 容器默认定义了一个 CMD 指令，使 nginx 在前台运行，但我们仍然可以在运行时定义一个不同的 CMD 指令，有需要时可以用来执行其他命令或启动一个 shell。

现在只需要更新我们的 Dockerfile，把这个新脚本用上：

```
FROM nginx:1.7

COPY default.conf /etc/nginx/conf.d/default.conf
COPY entrypoint.sh /entrypoint.sh

ENTRYPOINT ["/entrypoint.sh"]
CMD ["nginx", "-g", "daemon off;"] ❶
```

❶ 这个命令将会启动我们的代理，并在 docker run 没有指定任何命令的时候，作为参数传给我们的 entrypoint.sh 脚本。

把脚本设置成可执行文件，然后重建镜像。这一次，我们只称呼它 proxy，因为我们已经把有关 identidock 的信息全部剥离开来：

```
$ chmod +x entrypoint.sh
$ docker build -t proxy:1.0 .
...
```

要使用我们的新镜像，首先回到 identidock 文件夹，然后更新 prod.yml，使它能用上新镜像：

```
proxy:
  image: proxy:1.0
  links:
    - identidock
  ports:
   - "80:80"
```

```
    environment:
      - NGINX_HOST=45.55.251.164  ❶
      - NGINX_PROXY=http://identidock:9090
  identidock:
    image: amouat/identidock:1.0
    links:
     - dnmonster
     - redis
    environment:
      ENV: PROD
  dnmonster:
    image: amouat/dnmonster:1.0
  redis:
    image: redis:3
```

❶ 将此变量设置为你的主机 IP 地址或名称。

因此，如果现在把旧版本停掉，并重新启动应用程序，那么使用的将会是新的通用镜像。对于我们这个简单的 Web 应用，这些已经足够了，但由于我们使用了 Docker 连接，目前只能实现单一主机的配置。要是不采用更高级的网络连接和服务发现功能，我们将无法迁移到多主机架构（这对于容错和扩展是必须的），第 11 章和第 12 章将介绍如何使用这些功能。

当应用程序使用完毕，可以这样把它停止：

```
$ docker-compose -f prod.yml stop
...
$ docker-compose -f prod.yml rm
...
```

当你准备把云资源关掉时，只需这样做：

```
$ docker-machine stop identihost-do
$ docker-machine rm identihost-do
```

这时你应该到云服务提供商的网页界面上确认资源已被正确释放。

接下来看一些 Compose 的可替代方案。

设置 COMPOSE_FILE 变量

与其每次执行 Compose 的时候都指定 `-f prod.yml` 参数，不如通过设置 `COMPOSE_FILE` 环境变量达到同样目的。例如：

```
$ export COMPOSE_FILE=prod.yml
$ docker-compose up -d
...
```

它将会使用 prod.yml 文件，而不是默认的 docker-compose.yml。

强大的配置文件生成工具

将应用程序转化成 Docker 容器时，使用模板来生成配置文件是个很常用的手段，尤其是当程序本身并不支持环境变量时。当你的程序已经不像这里的范例那么简单的时候，为了避免由于正则表达式冲突而引起的奇怪错误，你需要认真考虑使用一个合适的模板处理器，例如 Jinja2 或 Go 模板。

由于这个问题很普遍，Jason Wilder 开发了一个工具，名叫 dockerize（https://github.com/jwilder/dockerize），目的是把这个过程自动化。Dockerize 通过一个模板文件以及环境变量来生成配置文件，然后调用原来的应用。这样做的话就可以用 dockerize 把 `CMD` 或 `ENTRYPOINT` 指令中的启动脚本封装起来。

然而，Jason 并不满足于此，他开发出了 docker-gen（https://github.com/jwilder/docker-gen）。docker-gen 可以利用容器的元数据（例如 IP 地址）和环境变量的值。它还可以保持在运行状态，随时响应 Docker 事件，例如，当有新容器被创建时，便会按当时的情况更新配置文件。一个很好的例子就是他的 nginx-proxy 容器，它会以 `VIRTUAL_HOST` 环境变量自动添加容器到负载均衡的组里。

9.3 执行选项

既然已经有了一个可以用于生产环境的系统，[2] 那又该如何在服务器上启动它呢？[3] 目前为止，我们已经看过 Compose 和 Machine 这两个项目，但它们都比较新，而且正在快速开发中，因此如果在生产环境中使用它们的话，就必须非常谨慎，除非是小的副项目（在写这部分的时候，Docker 网站上都有相关警告）。这两个项目成熟得很快，用于生产环境的功能正在迅速开发。如果想了解它们的发展方向，可以在 GitHub 仓库中找到它们的路线图，这对了解它们何时适合用于生产环境非常有用。

如此说来，如果 Compose 不适合，那又该用什么呢？让我们来看看其他的可能性。以下所有代码假设镜像都已经在 Docker Hub 上，而不是在服务器上创建的。如果你想跟着一起做，要么把你自己的镜像推送到寄存服务，要么使用我在 Docker Hub 上的镜像（`amouat/identidock:1.0`、`amouat/dnmonster:1.0` 和 `amouat/proxy:1.0`）。

9.3.1 shell脚本

不用 Compose 就能启动容器的最简单方法便是 shell 脚本，我们只需编写一个简短的 shell 脚本，它会执行 Docker 命令来启动容器。这样做足以应付大多数简单的用例，如果加上一些监控的命令，还可以确保当有问题发生并需要处理时，你能察觉得到。但长远来看，这个做法远非完美；随着时间发展，这个脚本会被不断加入其他功能，使得你将来很有可

注 2：其实，现在还不太适合用于生产环境。在公开你的应用给所有人访问之前，认真思考如何保证它的安全性是非常重要的。第 13 章会有更详细的介绍。

注 3：别忘了，你还要想想如何处理监控和日志的记录。请看第 10 章。

能不得不维护一个变得杂乱无章的脚本。

我们可以在执行 `docker run` 时使用 `--restart` 参数来确保不正常退出的容器能够自动重新启动。该参数指定容器重新启动的策略，它可以是 `no`、`on-failure` 或 `always`。默认值是 `no`，表示容器永远不会自动重启。`on-failure` 策略只会在容器的退出值为非 0 的时候尝试重启，还可以指定重试的最大次数（例如 `docker run --restart on-failure:5` 将尝试重启容器最多 5 次，之后便会放弃）。

下面的脚本（名为 deploy.sh）会启动并运行我们的 identidock 服务：

```
#!/bin/bash
set -e

echo "Starting identidock system"

docker run -d --restart=always --name redis redis:3
docker run -d --restart=always --name dnmonster amouat/dnmonster:1.0
docker run -d --restart=always \
  --link dnmonster:dnmonster \
  --link redis:redis \
  -e ENV=PROD \
  --name identidock amouat/identidock:1.0
docker run -d --restart=always \
  --name proxy \
  --link identidock:identidock \
  -p 80:80 \
  -e NGINX_HOST=45.55.251.164 \
  -e NGINX_PROXY=http://identidock:9090 \
  amouat/proxy:1.0

echo "Started"
```

注意，我们其实只是把 docker-compose.yml 文件转换成同样功能的 shell 命令。但与 Compose 不同，它并没有在失败后清理环境的逻辑操作，也没有检查是否已经有容器正在运行。

在 Digital Ocean 的情况下，我现在可以使用以下的 `ssh` 和 `scp` 命令，通过 shell 脚本来启动 identidock：

```
$ docker-machine scp deploy.sh identihost-do:~/deploy.sh
deploy.sh                                         100% 575     0.6KB/s   00:00
$ docker-machine ssh identihost-do
...
$ chmod +x deploy.sh
$ ./deploy.sh
Starting identidock system
3b390441b16eaece94df7e0e07d1edcb4c11ce7232108849d691d153330c6dfb
57459e4c0c2a75d2fbcef978aca9344d445693d2ad6d9efe70fe87bf5721a8f4
5da04a34302b400ec08e9a1d59c3baeec14e3e65473533c165203c189ad58364
d1839d8de1952fca5c41e0825ebb27384f35114574c20dd57f8ce718ed67e3f5
Started
```

其实也可以直接在 shell 中运行这些命令。选择使用脚本主要是出于方便注释和移植的考

虑，如果我想在一台新主机上启动 identidock，很容易就能找出用于启动相同版本应用的指令。

当需要更新镜像或对其进行更改时，可以利用 Machine 把本地的客户端连接到远程的 Docker 服务器，或者直接登入远程的服务器并使用它上面的客户端。如要做到零停机更新容器，你需要在容器前安装一个负载均衡器或反向代理，然后执行如下动作。

(1) 使用已更新的镜像启动一个新的容器（最好避免直接更新镜像）。
(2) 把负载均衡器指向新镜像，可以是部分流量或全部流量。
(3) 测试新容器并确保它工作正常。
(4) 把旧容器关掉。

另外，请参阅 8.6 节下的辅助栏“在生产环境中测试”，其中有几种如何在不影响服务的情况下部署更新的技巧。



重启时连接中断

旧版本的 Docker 曾经有容器重启时连接中断的问题。如果遇到类似问题，请确保运行的 Docker 是最新版本。本书写作之际，我的 Docker 版本是 1.8，它并没有这个问题；当容器的 IP 地址有任何改变的时候，它都会自动传达到已被连接的容器。另外请注意，在被连接的容器上，只有 /etc/hosts 将会被更新，而环境变量并不会。

下面几节将会介绍，如何利用一些你应该已经熟悉的技术来控制容器的启动和部署。第 12 章将会介绍一些专门设计给 Docker 的新工具，以及怎样通过它们来解决这个问题。

9.3.2 使用进程管理器（或用systemd控制所有进程）

除了依赖于 shell 脚本和 Docker 的重启功能，还可以使用进程管理器或 init 系统（如 systemd 或 upstart）来启动你的容器。如果你的主机有一些服务不是在容器中运行，而是依赖于一个或多个容器的话，那么这样做就特别有用。如果你想这样做，有一些问题你必须注意。

- 必须确保未使用 Docker 的自动重启容器功能，即执行 docker run 命令时不能使用 `--restart=always` 参数。
- 通常情况下，你的进程管理器将监控 `docker client` 客户端进程，而不是容器内的进程。这样做大部分时间都不会有问题，但如果网络连接中断或其他事情出错，Docker 客户端就会退出，但容器却仍在运行，这就可能导致问题发生。相反，如果进程管理器监视的是容器内的主进程，那么就会好很多。这个情况在未来可能会有所改变，但在那之前，你应该关注 systemd-docker 这个项目（https://github.com/ibuildthecloud/systemd-docker），它通过控制容器的 cgroup 来绕过这个问题。[有关该问题的更多信息，参见 GitHub 上的这一问题（https://github.com/docker/docker/issues/6791）。]

为了演示如何通过 systemd 来管理容器，下面的服务文件可以用于已采用 systemd 的主机上，用来启动我们的 identidock 服务。在这个例子中，我使用了 CentOS 7，但其他基于

systemd 的发行版的做法也大同小异。我并没有包括 upstart 的例子，因为似乎所有主流的发行版都正在往 systemd 迁移。所有文件都应放在 /etc/systemd/system/ 之下。

先来看看 Redis 容器的服务文件 identidock.redis.service，它并不依赖于任何其他的容器：

```
[Unit]
Description=Redis Container for Identidock
After=docker.service
Requires=docker.service ❶

[Service]
TimeoutStartSec=0 ❷
Restart=always
ExecStartPre=-/usr/bin/docker stop redis ❸
ExecStartPre=-/usr/bin/docker rm redis
ExecStartPre=/usr/bin/docker pull redis ❹
ExecStart=/usr/bin/docker run --rm --name redis redis

[Install]
WantedBy=multi-user.target
```

❶ 必须确保启动容器之前 Docker 已经运行。
❷ 由于 Docker 的命令可能需要一些时间来运行，把超时设定关闭将最省事。
❸ 在启动容器之前，先把名字相同的旧容器移除，这意味着在重启的时候，Redis 的缓存将会被彻底破坏。但在 identidock 的情况下，这不是个问题。在命令前加上 - 号的意思是告诉 systemd，即使命令返回一个非零的代码也不应该中止。
❹ 进行 pull 的动作，确保运行的是最新版本。

identidock 的服务 identidock.identidock.service 与之前的服务类似，区别是它需要依赖其他服务：

```
[Unit]
Description=identidock Container for Identidock
After=docker.service
Requires=docker.service
After=identidock.redis.service ❶
Requires=identidock.redis.service
After=identidock.dnmonster.service
Requires=identidock.dnmonster.service

[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker stop identidock
ExecStartPre=-/usr/bin/docker rm identidock
ExecStartPre=/usr/bin/docker pull amouat/identidock
ExecStart=/usr/bin/docker run --name identidock \
  --link dnmonster:dnmonster \
  --link redis:redis \
  -e ENV=PROD \
  amouat/identidock

[Install]
WantedBy=multi-user.target
```

❶ 除了 Docker，还需要声明 identidock 所依赖的其他容器，具体来说是 Redis 和 dnmonster 容器。需要同时使用 `After` 和 `Requires`，以避免竞态条件发生。

代理服务（称为 identidock.proxy.service）内容如下：

```
[Unit]
Description=Proxy Container for Identidock
After=docker.service
Requires=docker.service
Requires=identidock.identidock.service

[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker stop proxy
ExecStartPre=-/usr/bin/docker rm proxy
ExecStartPre=/usr/bin/docker pull amouat/proxy
ExecStart=/usr/bin/docker run --name proxy \
  --link identidock:identidock \
  -p 80:80 \
  -e NGINX_HOST=0.0.0.0 \
  -e NGINX_PROXY=http://identidock:9090 \
  amouat/proxy

[Install]
WantedBy=multi-user.target
```

最后是 dnmonster 服务（称为 identidock.dnmonster.service），内容如下：

```
[Unit]
Description=dnmonster Container for Identidock
After=docker.service
Requires=docker.service

[Service]
TimeoutStartSec=0
Restart=always
ExecStartPre=-/usr/bin/docker stop dnmonster
ExecStartPre=-/usr/bin/docker rm dnmonster
ExecStartPre=/usr/bin/docker pull amouat/dnmonster
ExecStart=/usr/bin/docker run --name dnmonster amouat/dnmonster

[Install]
WantedBy=multi-user.target
```

现在可以通过 `systemctl start identidock.*` 命令来启动 identidock 了。这个方法和 Docker 的重启功能之间的主要差异在于，重启一个已停止的容器将使 systemd 触发一连串的重启；假如 Redis 容器停止运行，identidock 和代理这两个容器也将重新启动。要是用 Docker 处理的话，情况就不一样了，因为它知道如何在不需要重新启动容器的情况下更新连接。

尽管存在前面所提到的问题，但值得注意的是，CoreOS 和 Giant Swarm 的 PaaS 服务都使用 systemd 来控制容器。目前，似乎可以客观地说，Docker 和 systemd 之间关系紧张并有待解决，因为两者都希望能负责管理主机上服务的生命周期。

9.3.3 使用配置管理工具

如果你的机构负责的主机数目较多，或许你已经在使用某种形式的配置管理（configuration management，CM）工具（如果你还未使用，或许应该考虑一下）。所有项目都需要考虑如何保证 Docker 主机上的操作系统是最新的，尤其是在安全补丁方面。然后，需要确保你正在运行的 Docker 镜像也是最新的，并且没有把不同版本的软件混着用。一些 CM 解决方案，诸如 Puppet、Chef、Ansible 和 Salt，都旨在帮助管理这些问题。

把 CM 工具应用在容器中有两大方法。

(1) 可以把容器当作虚拟器看待，让 CM 软件管理和更新容器内的软件。

(2) 可以让 CM 软件管理 Docker 主机上的容器，以确保容器运行的镜像版本正确无误，这样容器本身就会被视为可替换但不可修改的黑盒子。

第一种方法是可行的，但不符合 Docker 的哲学。这样做与 Dockerfile 冲突，也与 Docker 的“小容器且只有单一进程”的核心哲学背道而驰。在接下来的内容中，我们将重点关注第二种方法，因为它更贴近于 Docker 的理念和微服务架构的方向。

在这个方法中，容器本身好比虚拟机领域中的*黄金镜像*（golden image），它一旦运行起来就不会被修改。当你需要更新它的时候，要用一个运行新镜像的容器把它整个替换，而不是试图改变镜像中正在运行的东西。这样做有一个很大的好处，那就是只需查看镜像的标签，就能确切地知道容器中运行的是什么东西（假设你使用了一套合理的标签系统，并且标签不会重用）。

下面来看一个实际操作的例子。

Ansible

在这个例子中我们将使用 Ansible，因为它很受欢迎，又容易上手，而且是开源的。虽然如此，我并不是说它比其他工具更好或更差！

与许多其他的配置管理解决方案不同，Ansible（http://www.ansible.com/）并不需要在主机上安装代理。它主要依靠 SSH 来配置主机。

Ansible 有一个 Docker 模块，具有创建和编排容器的功能。你可以在 Dockerfiles 中使用 Ansible 来安装和配置软件，但现在只考虑使用 Ansible 通过我们的 identidock 镜像来建立一个虚拟机。由于只是在一台主机上运行，Ansible 的功能并没有被充分利用，但它可以演示 Ansible 和 Docker 的配合使用能达到很好的效果。

虽然也可以安装 Ansible 的客户端，我们最好直接使用 Docker Hub 上的 Ansible 客户端镜像。虽然官方镜像还未出来，但 `generik/ansible` 镜像对测试来说足矣。

首先创建一个 hosts 文件，包含希望由 Ansible 来管理的服务器列表。其中还必须包括你的远程主机或虚拟机的 IP 地址。

```
$ cat hosts
[identidock]
46.101.162.242
```

现在，我们需要创建一个用于安装 identidock 的“playbook”。以下面的内容创建一个名为

identidock.yml 的文件，如果你想使用自己的镜像，可以替换镜像名称：

```
---
- hosts: identidock
  sudo: yes
  tasks:
  - name: easy_install
    apt: pkg=python-setuptools
  - name: pip
    easy_install: name=pip
  - name: docker-py
    pip: name=docker-py
  - name: redis container
    docker:
      name: redis
      image: redis:3
      pull: always
      state: reloaded
      restart_policy: always
  - name: dnmonster container
    docker:
      name: dnmonster
      image: amouat/dnmonster:1.0
      pull: always
      state: reloaded
      restart_policy: always
  - name: identidock container
    docker:
      name: identidock
      image: amouat/identidock:1.0
      pull: always
      state: reloaded
      links:
        - "dnmonster:dnmonster"
        - "redis:redis"
      env:
        ENV: PROD
      restart_policy: always
  - name: proxy container
    docker:
      name: proxy
      image: amouat/proxy:1.0
      pull: always
      state: reloaded
      links:
        - "identidock:identidock"
      ports:
        - "80:80"
      env:
        NGINX_HOST: www.identidock.com
        NGINX_PROXY: http://identidock:9090
      restart_policy: always
```

大部分的配置与 Docker Compose 非常相似，但仍有几点需要注意。

- 为了能够使用 Ansible 的 Docker 模块，还需要在主机上安装 `docker-py`。这样就需要安装更多的 Python 依赖关系。
- 其中的 pull 变量决定什么时候检查 Docker 镜像的更新。把它设置为 `always` 的话，每次执行任务时便会检查镜像是否有新版本。
- 其中的 `state` 变量决定容器应处于什么状态。把它设置为 `reloaded` 的话，每当配置被改动时，容器将重启。

虽然还有很多配置选项可用，但这个配置文件的效果已经与本章中的其他配置很接近了。

接下来要做的就是运行 playbook：

```
$ docker run -it \
    -v ${HOME}/.ssh:/root/.ssh:ro \  ❶
    -v $(pwd)/identidock.yml:/ansible/identidock.yml \
    -v $(pwd)/hosts:/etc/ansible/hosts \
    --rm=true generik/ansible ansible-playbook identidock.yml

PLAY [identidock] *************************************************************

GATHERING FACTS ***************************************************************
The authenticity of host '46.101.41.99 (46.101.41.99)' can't be established.
ECDSA key fingerprint is SHA256:R0LfM7Kf3OgRmQmgxINko7SonsGAC0VJb27LTotGEds.
Are you sure you want to continue connecting (yes/no)? yes
Enter passphrase for key '/root/.ssh/id_rsa':
ok: [46.101.41.99]

TASK: [easy_install] **********************************************************
changed: [46.101.41.99]

TASK: [pip] *******************************************************************
changed: [46.101.41.99]

TASK: [docker-py] *************************************************************
changed: [46.101.41.99]

TASK: [redis container] *******************************************************
changed: [46.101.41.99]

TASK: [dnmonster container] ***************************************************
changed: [46.101.41.99]

TASK: [identidock container] **************************************************
changed: [46.101.41.99]

TASK: [proxy container] *******************************************************
changed: [46.101.41.99]

PLAY RECAP ********************************************************************
46.101.41.99               : ok=8    changed=7    unreachable=0    failed=0
$ curl 46.101.41.99
<html><head><title>Hello...
```

❶ 这个命令可以让容器使用 SSH 密钥对（SSH key pair）来访问远程服务器。

执行它需要一定的时间，因为 Ansible 需要把镜像下载下来。不过一旦完成，我们的 identidock 应用就会运行起来。

我们还没有触及 Anisble 真正强大的功能。它还可以做很多事情，尤其是在定义更新流程方面，可以进行滚动更新而不会破坏依赖关系或导致停机时间过长。

9.4　主机配置

目前为止，这一章假设容器是在 Digital Ocean 提供的原生 Docker droplet（Digital Ocean 的术语，意为预先配置的虚拟机，而在写作本书的时候，它是基于 Ubuntu 14.04 的）中运行的。但关于主机的操作系统和基础设施，也有许多不同的选择，各自有不同的利弊。特别是如果你负责使用企业的内部资源运行 Docker 的话，那就更应该谨慎考虑你的选择。

尽管可以在裸机上运行 Docker 主机（无论是在企业内部还是在云端），但目前最实际的选择还是使用虚拟机。大多数组织都已经拥有某种程度的虚拟机服务，可以用来为容器部署所需主机，并能提供用户之间的隔离性和安全性的有力保障。

9.4.1　选择操作系统

关于操作系统，目前已有多个选择，各自有不同的优缺点。如果你想运行一个小型到中型的应用程序，你可能会发现最简单的选择便是使用你所了解的——如果你通常使用 Ubuntu 或 Fedora，而你或你的组织对它都很熟悉，那就使用它吧（但要注意稍后讨论到的存储驱动问题）。但你要是打算运行一个非常庞大的应用程序或集群（数百个或数千个横跨多台主机的容器），建议你考虑更专业化的选择，例如 CoreOS、Project Atomic 或 RancherOS，以及将在第 12 章讨论的编排方案。

如果你是在云端的主机上运行，其中大部分都已经有随时可用的 Docker 镜像，而且已经在它们的基础设施上测试和验证过。

9.4.2　选择存储驱动程序

目前已有多个 Docker 支持的存储驱动程序，将来还会支持更多的驱动。选择合适的存储驱动程序对于确保生产环境中的可靠性和效率至关重要。哪个驱动程序最适合，取决于你的使用场景和运维经验。目前有下列这些选择。

AUFS

　　Docker 支持的第一个存储驱动程序。迄今为止，它大概是受过最多考验和最常用的驱动程序。与 Overlay 驱动相同，它的主要优势之一就是能够共享不同容器之间的内存页——如果两个容器从相同的底层数据层载入程序库或数据，那么操作系统能够让这两个容器使用相同的内存页。AUFS 的主要问题是它不在主线内核（mainline kernel）中，虽然它被 Debian 和 Ubuntu 采用已有一段时间。此外，AUFS 的操作是在文件级别，因此即使你对一个大文件只做了一个小小的改动，整个文件也都会被复制到容器的读 / 写层。与此相反，BTRFS 和 Device mapper 在块级别上操作，因此对于大文件而言更能有效地利用空间。如果你目前使用的是 Ubuntu 或 Debian 主机，那么你使用的驱动程序很

有可能就是 AUFS。

Overlay

与 AUFS 非常相似，它被合并到 Linux 内核的 3.18 版本。Overlay 很可能是日后主要支持的存储驱动，它的性能比 AUFS 稍好一些。目前，它主要的缺点是需要一个较新版本的内核（对大多数发行版而言，意味着需要对内核打补丁），并且它还没有像 AUFS 和一些其他选项那样经过那么多考验。

BTRFS

一个“写时复制”（copy-on-write）的文件系统[4]，专注于支持容错性、非常大的文件和卷（volume）。由于 BTRFS 有一些奇怪的行为和陷阱（特别是关于 chunk），因此只建议对 BTRFS 有经验的组织使用，或者当需要一些 BTRFS 特有而其他驱动程序并未支持的特性时。如果你的容器利用块级别的支持来读取和写入非常大的文件，它会是一个不错的选择。

ZFS

这个备受宠爱的文件系统原本由 Sun Microsystems 开发。它在许多方面与 BTRFS 很类似，但可以说它具有更好的性能和可靠性。在 Linux 上运行 ZFS 并不简单，由于许可证的问题，它不能被放进内核。[5] 因此，ZFS 很可能只会被拥有充分的 ZFS 经验的机构所采用。

Device mapper

红帽系统默认使用它。Device mapper 是个内核驱动，它被用作某些技术的基础，诸如 RAID、设备加密和快照。Docker 使用 Device mapper 的自动精简配置（thin provisioning，有时称为 thinp）[6] 来达到块级别而非文件级别的写时复制。“thin pool”从稀疏文件（sparse file）分配，默认大小为 100GB。容器被创建的时候，将会被分配一个利用这个 pool 所建立的文件系统，其大小默认为 100GB（适用于 Docker 1.8）。由于是稀疏文件，实际磁盘的使用空间要少得多，但一个容器的使用空间不能超过 100GB，除非修改默认值。Device mapper 可以说是 Docker 的众多存储驱动中最为复杂的一个，也是问题和寻求帮助的一个主要来源。如果可能的话，我会建议使用其他驱动。但如果你真的要使用 Device mapper 的话，请记住它有很多选项可供调整，以提供更好的效能（尤其是把存储从“loopback”设备移到一个真正的设备，这是个不错的主意）。

VFS

默认的 Linux 虚拟文件系统。它没有实现写时复制，因此启动一个新容器时需要复制整个镜像。这样就会使启动容器变得很慢，并且大幅度增加了磁盘所需的空间。它的优点

注 4：不要问我 BTRFS 怎么念：有人念“ButterFS”，也有人念“BetterFS”；我念成“FSCK”。

注 5：Ubuntu 决定从 Ubuntu 16.04 开始把 ZFS 合并到它的内核中一同发布。——译者注

注 6：在自动精简配置中，客户端请求的资源不会立即全部分配完成，而只会在真正需要的时候分配。与此相反，传统密集配置（thick provisioning）则会立即预留客户端请求的资源，即使客户端也许仅使用资源的一小部分。

是非常简单，而且不需要任何特别的内核功能。如果你在使用其他的驱动时遇到问题，而且不介意 VFS 较差的性能的话（譬如你只有少数长时间使用的容器），那么它会是个合理的选择。

除非你有特别的理由去选择其他的驱动，我会建议你使用 AUFS 或 Overlay，即使这意味着需要更新内核。

切换存储驱动程序

假如你已经安装好所需的依赖关系，切换存储驱动程序是很容易的。只需重新启动 Docker 守护进程，并把适当的值传给 `--storage-driver` 参数（简写为 `-s`）。例如，如果你的内核支持 Overlay，你可以通过 `docker daemon -s overlay` 命令以 Overlay 存储驱动来启动 docker 守护进程。特别要注意 `--graph` 或 `-g` 参数，这个参数设置 Docker 运行时使用的根目录——你可能需要移动到适当的文件系统所在的分区来执行 Docker（例如使用 BTRFS 驱动的话，需要执行 `docker daemon -s btrfs -g /mnt/btrfs_partition`）。

要把改动固定为永久，需要编辑 Docker 服务的启动脚本或配置文件。如果是在 Ubuntu 14.04 上，那么需要编辑 /etc/default/docker 文件中的 `DOCKER_OPTS` 变量。

在存储驱动之间移动镜像

当切换存储驱动程序之后，你将无法访问所有的旧容器和旧镜像。回到之前的存储驱动就能恢复原状。要将镜像迁移到新的存储驱动程序，只需将镜像保存到一个 TAR 文件，然后在新文件系统里加载它。例如：

```
$ docker save -o /tmp/debian.tar debian:wheezy
$ sudo stop docker
$ docker daemon -s vfs
...
```

在新的终端下执行：

```
$ docker images
REPOSITORY  TAG    IMAGE ID      CREATED     VIRTUAL SIZE
$ docker load -i /tmp/debian.tar
$ docker images
REPOSITORY  TAG    IMAGE ID      CREATED     VIRTUAL SIZE
debian      wheezy b3d362b23ec1  2 days ago  84.96 MB
```

9.5 专门的托管方案

市面上已经有一些无需你直接管理主机的专门的容器托管方案。

9.5.1 Triton

Joyent 公司（https://www.joyent.com）的 Triton 也许是所有方案中最有趣的一个，因为它的内部并不使用虚拟机。这使得 Triton 相比于其他基于虚拟机的方案具有显著的性能优势，并且允许按单个容器部署。

Triton 没有使用 Docker 引擎，而是通过 Linux 虚拟化技术自我实现了一个基于 SmartOS 虚拟机管理程序（它的根源可以追溯至 Solaris）的容器引擎。通过实现 Docker 的远程 API，Triton 与常规的 Docker 客户端完全兼容，而 Docker 客户端被用作 Triton 的标准接口。Docker Hub 上的镜像也能正常使用。

Triton 是开源的，并提供两种托管版本，一种运行在 Joyent 云端，另一种是企业内部的版本。通过 Joyent 的公有云，我们可以迅速地把 identidock 运行起来。在 Triton 创建账户并把 Docker 客户端指向 Triton 后，尝试执行 `docker info`：

```
$ docker info
Containers: 0
Images: 0
Storage Driver: sdc
 SDCAccount: amouat
Execution Driver: sdc-0.3.0
Logging Driver: json-file
Kernel Version: 3.12.0-1-amd64
Operating System: SmartDataCenter
CPUs: 0
Total Memory: 0 B
Name: us-east-1
ID: 92b0cf3a-82c8-4bf2-8b74-836d1dd61003
Username: amouat
Registry: https://index.docker.io/v1/
```

注意操作系统（operating system）和执行驱动（execution driver）的值，它们表示我们不是运行在一个普通的 Docker 引擎上。可以使用 Compose 及以下的 triton.yml 文件启动 identidock，因为 Triton 支持大部分的 Docker 引擎 API：

```
proxy:
  image: amouat/proxy:1.0
  links:
    - identidock
  ports:
   - "80:80"
  environment:
    - NGINX_HOST=www.identidock.com
    - NGINX_PROXY=http://identidock:9090
  mem_limit: "128M"
identidock:
  image: amouat/identidock:1.0
  links:
   - dnmonster
   - redis
  environment:
    ENV: PROD
  mem_limit: "128M"
dnmonster:
  image: amouat/dnmonster:1.0
  mem_limit: "128M"
redis:
  image: redis
  mem_limit: "128M"
```

这个文件与 prod.yml 几乎一样，不同的是添加了一些关于内存的设置，用于告诉 Triton 容器的大小。我们不会构建自己的镜像，而是利用公共镜像（目前 Triton 还不支持 `docker build`）。

启动应用：

```
$ docker-compose -f triton.yml up -d
...
Creating triton_proxy_1...
$ docker inspect -f {{.NetworkSettings.IPAddress}} triton_proxy_1
165.225.128.41
$ curl 165.225.128.41
<html><head><title>Hello...
```

当发现一个需要发布的端口时，Triton 会自动使用能被公开访问的 IP。

当 Triton 上的容器使用完毕后，务必把它们停止并删除；Triton 会对已停止但未删除的容器收费。

通过原生的 Docker 工具来使用 Triton 服务是个很棒的经验，但也有一些瑕疵；并非所有的 API 接口都能支持，Compose 如何处理数据卷也有一些问题，但这些问题应该会在将来得到解决。

直到主流的云提供商相信 Linux 内核提供的隔离保证足够强大，运行容器时不用担心任何安全问题的那一天到来之前，Triton 仍然是运行容器化系统最具吸引力的解决方案之一。

9.5.2 谷歌容器引擎

谷歌容器引擎（GKE，https://cloud.google.com/container-engine/）建立在 Kubernetes 编排系统的基础上，运行容器的方式独树一帜。

Kubernetes 是一个由谷歌设计的开源项目，设计时吸取了他们曾在内部运行 Borg 集群管理器时的教训。[7]

部署应用到 GKE 时需要对 Kubernetes 有一些基本认识，并需要创建一些 Kubernetes 的配置文件，12.1.3 节中将会谈及这些。

虽然要付出更多的精力来配置应用，但回报是获得诸如自动复制及负载均衡这些服务。它们听起来好像只有那些高流量和含有大量分布式子系统的大型服务才需要的，但在你的服务需要保证正常运行时间的情况下，这些服务立刻就变得很重要了。

我强烈推荐使用 Kubernetes 来部署容器系统，尤其是 GKE，但必须注意的是，使用它们之后，你就会被绑定在 Kubernetes 的模式，迁移到其他供应商将会变得更困难。

注 7：“谷歌利用 Borg 实现大型集群管理”（“Large-scale cluster management at Google with Borg”，https://research.google.com/pubs/pub43438.html）这篇论文讲述了如何运行一个处理成千上万个任务的集群，非常精彩。

9.5.3 亚马逊EC2容器服务

亚马逊 EC2 容器服务（ECS，https://aws.amazon.com/ecs/）可以让你在亚马逊的 EC2 设施上运行容器。ECS 提供了一个 Web 界面，以及用于启动容器和管理底层 EC2 集群的 API。

在集群中的每个节点上，ECS 都会启动一个容器代理（container agent），它与 ECS 服务进行通信，并负责启动、停止和监视容器。

虽然在 ECS 上将 identidock 运行起来，会涉及 AWS 独有的界面以及几十个配置选项，但相对而言还是比较快的。在 ECS 注册并创建一个集群之后，我们需要为 identidock 上传一个"任务定义"（task definition）。下面的 JSON 文件可以用作 identidock 的定义。

```
{
  "family": "identidock",
  "containerDefinitions": [
    {
      "name": "proxy",
      "image": "amouat/proxy:1.0",
      "cpu": 100,
      "memory": 100,
      "environment": [
        {
          "name": "NGINX_HOST",
          "value": "www.identidock.com"
        },
        {
          "name": "NGINX_PROXY",
          "value": "http://identidock:9090"
        }
      ],
      "portMappings": [
        {
          "hostPort": 80,
          "containerPort": 80,
          "protocol": "tcp"
        }
      ],
      "links": [
        "identidock"
      ],
      "essential": true
    },
    {
      "name": "identidock",
      "image": "amouat/identidock:1.0",
      "cpu": 100,
      "memory": 100,
      "environment": [
        {
          "name": "ENV",
          "value": "PROD"
        }
      ],
      "links": [
```

```
          "dnmonster",
          "redis"
        ],
        "essential": true
      },
      {
        "name": "dnmonster",
        "image": "amouat/dnmonster:1.0",
        "cpu": 100,
        "memory": 100,
        "essential": true
      },
      {
        "name": "redis",
        "image": "redis:3",
        "cpu": 100,
        "memory": 100,
        "essential": false
      }
    ]
  }
```

每个容器需要指定内存大小（以 MB 为单位）和 CPU 单元数。`essential` 关键字定义当容器无法运行时任务是否应该停止。在我们的例子中，Redis 容器可以定义为非 essential，因为即使没有它，应用程序仍然能够工作。其他字段的含义看字面应该就能明白。

一旦任务创建成功，你便需要在集群上启动它。identidock 应启动为服务，而不是一次性任务。以服务运行意味着 ECS 将监控容器以确保它的可用性，并提供连接到亚马逊的弹性负载均衡（Elastic Load Balancer），把流量平均分布至各实例。创建服务时，ECS 会询问一个名字和它需要确保运行的任务数。创建服务后，等待任务启动时，你应该能够通过 EC2 实例的 IP 地址来访问 identidock。IP 地址可以在任务实例的详细信息页面上关于代理容器的扩展信息中找到。

停止服务和相关资源需要几个步骤。首先需要把服务的任务数量改为 0，以避免 ECS 在关闭任务的同时启动替代它们的任务。这个时候，你可以把服务删除。在集群能够被删除之前，还需要注销容器示例。还要注意曾经启动过的任何相关资源也必须停掉，如弹性负载均衡器或弹性块存储（Elastic Block Store，EBS）。

ECS 的幕后其实进行着非常多的技术工作。只需简单点击几下，便能启动数百甚至数千个容器，为你提供真正的扩展能力。容器部署到主机的调度算法是高度可配置的，允许用户按照自己的需求进行优化，如效率最大化或可靠性最大化。用户可以把默认的 ECS 调度程序替换成自己的方案，或者第三方方案，如 Marathon（参见 12.1.4 节）。

ECS 还集成了现有的亚马逊功能，譬如能够把负载分布到多个实例的弹性负载均衡，以及实现持久存储的弹性块存储。

9.5.4　Giant Swarm

Giant Swarm（https://giantswarm.io/）自称是“独树一帜的微服务架构解决方案”，意思是通过它专门的配置格式，启动基于 Docker 的系统既快速又简便。Giant Swarm 提供两种不

同的方案，一种是共享集群上的托管方案，另一种是独立方案（Giant Swarm 将为你部署和维护裸机上的主机），也是部署在企业内部的解决方案。在撰写本书的时候，共享方案仍处于 Alpha 阶段，独立方案则已经可以用于生产环境。

Giant Swarm 不常用到虚拟机，甚至不会使用，这种方式较为罕见。有严格安全要求的用户使用独立的裸机作为主机，但在共享集群上，不同用户的容器运行则是彼此相邻的。

接下来看看如何在 Giant Swarm 的共享集群上运行 identidock。假设你已取得 Giant Swarm 的访问权限，并已安装它的 swarm 命令行工具，[8] 现在就开始创建下面的配置文件，并把它保存为 swarm.json：

```
{
    "name": "identidock_svc",
    "components": {
        "proxy": {
            "image": "amouat/proxy:1.0",
            "ports": [80],
            "env": {
                "NGINX_HOST": "$domain",
                "NGINX_PROXY": "http://identidock:9090"
            },
            "links": [ {
                "component": "identidock",
                "target_port": 9090
            }],
            "domains": { "80": "$domain" }
        },
        "identidock": {
            "image": "amouat/identidock:1.0",
            "ports": [9090],
            "links": [
            {
                "component": "dnmonster",
                "target_port": 8080
            },
            {
                "component": "redis",
                "target_port": 6379
            }
            ]
        },
        "redis": {
            "image": "redis:3",
            "ports": [6379]
        },
        "dnmonster": {
            "image": "amouat/dnmonster:1.0",
            "ports": [8080]
        }
    }
}
```

注 8：虽然 Docker 的集群方案也叫 swarm，但二者彼此没有任何关系。

现在是时候让 identidock 登场了。

```
$ swarm up --var=domain=identidock-$(swarm user).gigantic.io
Starting service identidock_svc...
Service identidock_svc is up.
You can see all components using this command:

    swarm status identidock_svc

$ swarm status identidock_svc
Service identidock_svc is up

component   image                  instanceid    created              status
dnmonster   amouat/dnmonster:1.0   m6eyoilfiei1  2015-09-04 09:50:40  up
identidock  amouat/identidock:1.0  r22ut7h0vx39  2015-09-04 09:50:40  up
proxy       amouat/proxy:1.0       6dr38cmrg3nx  2015-09-04 09:50:40  up
redis       redis:3                jvcf15d6lpz4  2015-09-04 09:50:40  up
$ curl identidock-amouat.gigantic.io
<html><head><title>Hello...
```

这里展示了其中一个 Giant Swarm 的配置文件中 Docker Compose 所不具备的特色，那就是允许使用模板变量的能力。在刚才的例子中，我们在命令行上传入主机名，swarm 便将 swarm.json 中的 `$domain` 替换为主机名的值。其他 swarm.json 提供的功能包括可以定义 pod，即一组能同时调度的容器，以及定义某个容器允许运行的实例数量。

最后，除了 swarm 的命令行工具，它还有一个用于监控服务和查看日志的 Web 用户界面，以及可用于实现 Giant Swarm 自动化的 REST API。

9.6 持久性数据和生产环境容器

可以说，在 Docker 的世界里，有关数据存储的理论并没有改变多少，至少在规模较大的系统中是如此。如果你运行自己的数据库，你可以选择使用 Docker 容器、虚拟机或裸机。如果你有大量数据，由于数据迁移的难度，实际上你的虚拟机或容器最终只会被固定在它的主机上。这意味着通常与容器挂钩的可移植性这个优点不再适用，但你可能仍然会为了保持平台的一致性以及隔离的好处而继续使用容器。如果你对性能有所顾虑，可以使用 `--net=host` 和 `--privileged` 这两个参数，以确保容器实际上能与主机的虚拟机或裸机一样高效，但要注意安全方面它们可能带来的后果。如果你没有运行自己的数据库，而是使用如 Amazon RDS 的托管服务，那么就没有什么需要特别注意的。

而当规模较小，即容器使用的配置文件以及数据量不太多时，你可能会觉得数据卷限制颇多，因为它会把你绑定到某一台主机上，使得扩展和迁移容器更加困难。你可以考虑将这些数据迁移到单独的键值存储或数据库，毕竟它们也可以在容器中运行。一个挺有意思的另类方法是使用 Flocker（https://github.com/ClusterHQ/flocker）来管理你的数据卷。Flocker 利用 ZFS 文件系统的功能来支持容器数据的迁移。如果你正尝试采用微服务架构，那么你会发现，尽可能使容器保持无状态（stateless）将会令很多事情变得简单。

9.7 分享秘密信息

你可能有一些敏感数据，例如密码和 API 密钥，需要在安全的情况下与你的容器共享。下面将会介绍这样做的几种方法，以及它们各自的优缺点。[9]

9.7.1 在镜像中保存秘密信息

千万不要这样做，这个主意糟透了。™

虽然这可能是最简单的方法，但它意味着任何能够获取镜像的人都能获取你的秘密信息。而且它也无法被删除，因为它会在先前的镜像层中一直存在。即使你使用的是私有寄存服务，或没有使用任何寄存服务，也难免遇到别人不小心把镜像公开的情况，而且别人也不一定要获取到镜像才能得知这些秘密信息。此外，这样做将使你的镜像只能用于特定的配置。

你可以把秘密信息进行加密，然后存储到镜像中，但你仍然需要想办法将解密密钥传到使用者的手中，而这会让攻击者有机可乘，可说是多此一举。

我建议还是干脆打消这个念头吧。我把这个“方法”提出来的原因是，当有人真的这样做了而且出了大事之后，我就可以让大家看看，其实我早就在此给你们提过醒了。

9.7.2 通过环境变量传递密钥

通过环境变量传递密钥是个非常简单直接的方法，和把密钥写到镜像中相比要好很多。它实现起来也很简单，只需把密钥作为参数传给 `docker run`。举例如下：

```
$ docker run -d -e API_TOKEN=my_secret_token myimage
```

很多应用程序和配置文件都能直接支持使用环境变量。但对于未能直接支持的，你可能需要实现类似 9.2 节中所写的脚本。

这个方法受到备受欢迎和推崇的“十二要素应用宣言”（The Twelve-Factor App，http://12factor.net）的推荐，它是一种构建软件即服务的应用的方法论。[10] 虽然我强烈建议大家阅读这个文档并运用其中的大部分建议，但在环境变量中存储密钥还是有一些严重的缺陷，列举如下。

- 环境变量对所有子进程、docker inspect 以及任何连接容器皆可见。它们都不具备能看见密钥的充分理由。
- 环境变量通常会被保存下来，以供日志和调试之用。这使得在调试日志和问题跟踪系统中暴露密钥的风险极大。
- 它们不能被删除。理想情况下，当我们使用密钥之后，我们希望能够把它覆盖或彻底删除，但对 Docker 容器来说是不可能做到的。

基于上述原因，我不建议使用这个方法。

注 9：如果你使用了配置管理软件（如 Ansible）来管理你配置的容器的话，那么它可能已经实现了相关的解决方案，或者能提供解决这一问题的相关方法。

注 10：值得指出的是，“十二要素应用宣言”的出现要早于 Docker 容器，因此某些建议需要经过调整才能适用。

9.7.3 通过数据卷传递密钥

使用数据卷来分享密钥是稍微好一点的解决方案，虽然远非完美。举例如下：

```
$ docker run -d -v $(pwd):/secret-file:/secret-file:ro myimage
```

除非你映射的配置文件整个都是密钥，否则你很可能需要制作脚本来处理用这个方法传递的密钥。如果你感觉自己技术了得，还可以创建一个临时文件存放密钥，并在读完它之后把文件删除（但小心，千万不要把原始文件删除）。

对于使用环境变量的配置文件，还可以创建一个脚本来设置环境变量，并在运行相应的应用程序前对该脚本进行 source 操作来读取它的内容。举例如下：

```
$ cat /secret/env.sh
export DB_PASSWORD=s3cr3t
$ source /secret/env.sh && run_my_app.sh
...
```

这样做的一个主要优点是，变量不会被暴露给 `docker inspect` 或连接容器。

这个方法的主要缺点是，密钥需要以文件的形式保存，这样就很容易被提交到版本控制系统。并且由于通常需要创建脚本，这个方案也可能会更繁琐。

9.7.4 使用键值存储

最好的解决办法可以说是使用键值存储来保存密钥，在运行时需要获取密钥的时候便从容器中读取它们。这样做可以对密钥多加一重控制，这是前文所述的方法做不到的，不过需要更多的配置步骤，并且要信任你所使用的键值存储。

以下几种解决方案都能提供这方面的功能。

KeyWhiz（https://square.github.io/keywhiz/）

　　将密钥加密并存放在内存中，提供 REST API 和命令行界面（CLI）两种访问接口。由 Square 公司（一家支付处理公司）开发及使用。

Vault（https://hashicorp.com/blog/vault.html）

　　可以把已加密的密钥存储在多种后端中，包括文件和 Consul。它也提供 CLI 和 API。虽然它还有几个 KeyWhiz 没有的功能，但我认为它们还不太成熟。Vault 由 Hashicorp 公司开发，它同时也是服务发现工具 Consul 和基础设施配置工具 Terraform 背后的公司。

Crypt（https://xordataexchange.github.io/crypt/）

　　它把已加密的值保存在 etcd 或 Consul 键值存储中。这种方法最大的好处是对密钥的控制程度提升到了前所未有的高度。它使得更改和删除密钥更容易，还可以给密钥附上“租期”，使它们在规定期限之后失效，或者在出现安全警报时封锁对密钥的访问。

然而还有一个问题：键值存储如何对容器进行身份验证？按照一般的做法，仍然需要把私钥以数据卷的方式传给容器，或者以环境变量的方式把令牌传给它。先前已说过反对使用环境变量，这里可以用*一次性令牌*来解决，它在使用后便会立即失效。目前正在开发的另

一种解决方案是通过对键值存储使用数据卷插件，插件把键值存储中的秘密信息以文件的形式挂载到容器内。GitHub 上（https://github.com/calavera/docker-volume-keywhiz）有更多关于如何把这种方法应用于 KeyWhiz 存储的详细信息。

这种解决方案将会是未来的方向。尽管实施这种方案时要面对很多难题，但它能够对敏感数据提供的控制精度却是非常值得去实现的，而且随着工具的改良，实施的难度也将降低。不过你不妨多等一会儿，看看它们以后的发展情况再作决定。当前你可以先用数据卷来传递私密信息，但要非常小心，千万不要把它们提交到版本控制系统中。

9.8 网络连接

第 11 章将会深入讨论网络连接。不过值得注意的是，如果你使用的是原生的 Docker 网络连接技术，系统性能将会大打折扣——建立 Docker bridge 和使用 veth[11] 意味着大量的网络路由发生在用户空间，这要比硬件的路由器或内核处理要慢得多。

9.9 生产环境的寄存服务

在 identidock 的范例中，我们一直只使用 Docker Hub 来获取我们的镜像。大多数用于生产环境的设置都会包含一个（或多个）寄存服务器，用于提供镜像的快速存取，并避免将这个关键的基础设施依赖于第三方（有些机构不放心把他们的代码存储在第三方，无论是否在私人仓库里）。有关设置寄存服务的详细信息，可以参考前面的 7.4.1 节。

你必须保持寄存服务器中的镜像是最新的并且是正确的，这件事情非常重要，因为你绝对不会希望主机能够下载镜像过去的版本，甚至存在安全风险的镜像。出于这个原因，对寄存服务器进行定期审核是个很好的主意，关于这部分的内容将会在 13.10 节中讲述。但请谨记，每个 Docker 主机都有它自己的镜像缓存，审核时千万不要忽略它。

镜像的复制（mirroring）、支持可扩展的拓扑结构的类似用途以及高可用性，目前都正由 Docker 分布项目（https://github.com/docker/distribution/）进行开发。

9.10 持续部署/交付

持续交付是把持续集成延伸到生产环境的一套方法；理论上工程师能够在开发环境中修改程序并对它们进行测试，并且只需简单点击一个按钮，就可以把应用准备好，用于正式部署。持续部署更进一步，把能够通过测试的改动自动推送到正式部署的环境。

第 8 章介绍过如何使用 Jenkins 建立一个持续集成系统。扩展到持续部署可以通过把镜像推送到生产环境的寄存服务器，以及将运行中的容器迁移到新镜像来实现。无需停机的镜像迁移需要先启动新容器，并给流量重新配置路由，然后才可以停止旧容器。有几种可行的方法来安全地实现它，例如在 8.6 节下的辅助栏“在生产环境中测试”中曾经提过的蓝 /

注 11：虚拟以太网（Virtual Ethernet），又称为 veth，是一个拥有自己 MAC 地址的虚拟网络设备，为了在虚拟机中使用而开发。

绿部署（blue/green deployment）和渐增式部署（ramped deployment）。实施这些技术通常会用到内部开发的工具，虽然类似 Kubernetes 的框架提供了内置的解决方案，我还是期待将来能在市场上看到一些专门的工具。

9.11 总结

这一章的内容非常多，覆盖了关于部署容器到生产环境时需要考虑的方方面面，即使 identidock 这样简单的程序也是如此。

虽然容器这个领域还很新，但目前已经有一些生产级别的托管容器的方案可供选择。究竟哪个是最好的选择，取决于你的系统规模和复杂程度，以及你愿意花费多少精力和金钱在部署和维护上。小型的部署只需在云端的虚拟机上运行 Docker 引擎来进行管理，但是对于较大型的部署，维护的负担则会很大。这个问题可以通过将于第 12 章讨论的系统（比如 Kubernetes 和 Mesos）来缓解，或者通过使用专门的托管服务，比如 Giant Swarm、Triton 或 ECS。

本章中涉及了一些在生产环境中经常遇到的问题，从启动容器这样简单的问题，到如何传递秘密信息、处理数据卷，以至持续部署这样棘手的问题。其中的一些问题需要专门为容器化系统制定的新方法，特别是当系统由动态微服务所组成时。为了解决这些问题，人们将会开发新的模式并寻找最佳实践，这些实践将发展为新的工具和框架。现在容器已经可以在生产环境中稳定使用，但未来会更加光明。

第10章

日志记录和监控

如果想保持任何一个稍微复杂一点的系统能够正常运作，并能有效地对问题进行调试，那么对容器进行有效的监控和日志记录是必不可少的。在微服务架构里，日志和监控将更为重要，因为机器的数量变得更多。由于容器的短暂生命周期特性，当你调试问题的时候，那个容器可能已经不存在了，这使得日志集中管理成为不可或缺的工具。

在最近的几个星期乃至几个月，日志记录和监控的解决方案数量激增。现有的监控和日志记录的供应商已经开始提供专门的容器解决和集成方案。本章会尽量介绍各种已有的选项和技术，并重点推介免费和开源的产品。我们将会看到如何扩展 identidock 应用至日志记录和监控方案，而这些方案还可以很容易地扩展到更大型的应用中。

这一章的代码已发布到 GitHub 仓库（https://github.com/using-docker/logging）上。与第 9 章一样，范例中使用的镜像均来自 Docker Hub，如果你愿意，也可以用自己的镜像来替换 identidock 容器。

可以使用 `v0` 标签获取这一章开始时的代码：

```
$ git clone -b v0 \
  https://github.com/using-docker/logging/
...
```

后续的标签代表代码在本章中演变的各个阶段。

除此以外，也可以在 GitHub 项目的 Releases 页面（https://github.com/using-docker/logging/releases）下载各个标签的代码。

10.1 日志记录

首先来看一下 Docker 默认的日志记录是如何工作的，然后我们会尝试将一个完整的日志记录解决方案集成到 identidock 中，最后会看看有哪些可选的解决方案，以及生产环境中需要考虑到的问题。

10.1.1 Docker默认的日志记录

先从最简单的方案说起，那就是 Docker 自带的方案。如果没有指定任何参数或安装任何日志软件，Docker 将记录所有发送到 `STDOUT` 和 `STDERR` 的信息。之后日志可以通过 `docker logs` 命令取得。举例如下：

```
$ docker run --name logtest debian sh -c 'echo "stdout"; echo "stderr" >&2'
stderr
stdout
$ docker logs logtest
stderr
stdout
```

也可以使用 `-t` 参数来获取时间戳：

```
$ docker logs -t logtest
2015-04-27T10:30:54.002057314Z stderr
2015-04-27T10:30:54.005335068Z stdout
```

还可以使用 `-f` 参数，让正在运行的容器不停输出日志：

```
$ docker run -d --name streamtest debian \
             sh -c 'while true; do echo "tick"; sleep 1; done;'
13aa6ee6406a998350781f994b23ce69ed6c38daa69c2c83263c863337a38ef9
$ docker logs -f streamtest
tick
tick
tick
tick
tick
tick
...
```

还可以利用 Docker 的远程 API 达到同样效果，[1] 它让使用程序对日志的转发和处理操作变得可能。如果你正使用 Docker Machine，可以执行如下操作：

```
$ curl -i --cacert ~/.docker/machine/certs/ca.pem \
          --cert ~/.docker/machine/certs/ca.pem \
          --key ~/.docker/machine/certs/key.pem \
          "https://$(docker-machine ip default):2376/containers/\
$(docker ps -lq)/logs?stderr=1&stdout=1"
tick
```

注 1：有关远程 API 的更多信息以及如何启用它，可以查看官方文档（https://docs.docker.com/engine/reference/api/docker_remote_api/）。

```
tick
tick
...
```

如果你使用的是 Mac OS，那么要注意 curl 的行为略有不同，而且你需要创建一个同时包含 ca.pem 和 key.pem 的证书。关于这方面的更多细节参见 Open Solitude 博客（http://opensolitude.com/2015/07/12/curl-docker-remote-api-os-x.html）。

默认的日志记录有一些缺点。它只能处理 `STDOUT` 和 `STDERR`，如果你的应用只把日志记录到文件中的话，那么就有问题了。它也不懂得清理日志文件，这意味着如果你尝试使用如 `yes`（它只会不断重复打印“yes”到 `STDOUT`）的程序来保持容器运行，你很快就会发现容器把你的磁盘上所有可用空间用尽。[2] 例如：

```
$ docker run -d debian yes
ba054389b7266da0aa4e42300d46e9ce529e05fc4146fea2dff92cf6027ed0c7
```

在执行 `docker run` 时，可以提供 `--log-driver` 参数来选择不同的日志记录方法。默认的日志记录方法可以在启动 Docker 守护进程时通过 `--log-driver` 参数变更。可选的日志记录方法有下列几种。

`json-file`

　这是我们刚刚看过的默认记录方法。

`syslog`

　syslog 系统日志驱动，接下来便会介绍它。

`journald`

　这个驱动使用的是 systemd 的 journal 日志。

`gelf`

　Graylog Extended Log Format（GELF）驱动。

`fluentd`

　将日志信息转发到 fluentd（http://www.fluentd.org/）。

`none`

　关闭日志记录。

在某些情况下，例如之前提到的 `yes` 例子，关闭日志将非常有用。

10.1.2　日志汇总

不管使用哪种日志驱动，它只能解决部分需求，尤其是对于多主机的系统。我们想要做的是把所有的日志汇总到同一个地方（日志有可能是跨主机的），这样能够利用工具对日志

注 2：是的，我愚蠢到碰过这个钉子才学会。

进行分析和监控。

有两种基本方法可以用来达到这个目的。

(1) 在所有的容器内运行多一个进程，这个进程的角色相当于把日志转发到汇总服务的代理。
(2) 在主机上或在另一个独立的容器中，收集日志并转发到汇总服务。

第一种方法是可行的，有时候还会被用上，但它会使镜像变大，还会增加多余的运行进程，因此这里只会考虑第二种方法。

从主机访问容器日志有几种方法。

(1) 可以通过 Docker API 以编程的方式访问日志。优点是这是官方所支持的，缺点是 HTTP 连接将导致额外的开销。下一节将介绍如何使用 Logspout 来这样做。
(2) 如果使用 syslog 系统日志驱动，就可以利用 syslog 自身的功能来自动转发日志，在 10.1.4 节下的辅助栏“使用 rsyslog 转发日志”中将有所介绍。
(3) 可以直接从 Docker 目录访问日志文件。这个做法将在 10.1.5 节中介绍。

如果你使用的应用程序不能把日志写到 `STDOUT` 或 `STDERR`，而是坚持只记录到文件中，可以参考 10.1.3 节的说明文字“如何处理把日志记录到文件的应用”中提到的变通方法。

10.1.3　使用ELK进行日志记录

要把日志记录的功能添加到我们的 identidock 应用中，我们将使用有时候被称为 ELK 的组合，ELK 是 Elasticsearch、Logstash 和 Kibana 的首字母缩写。

Elasticsearch（https://github.com/elastic/elasticsearch）

　　接近实时搜索的文本搜索引擎。为了能够应付大量数据，它的设计可以很轻松地扩展到多个节点，非常适合在海量日志数据中进行搜索。

Logstash（https://github.com/elastic/logstash）

　　这个工具适合读取原始日志，然后对其解析和过滤，再把结果发送到其他服务，例如索引或存储（在我们的例子中，Logstash 会把结果发送到 Elasticsearch）。它广泛支持不同的输入和输出类型，并备有适用于各式各样的应用程序日志的解析器。

Kibana（https://github.com/elastic/kibana）

　　基于 JavaScript 的 Elasticsearch 图形界面。它可以用来运行 Elasticsearch 的查询，并以各种图表的形式显示结果。它还提供了仪表盘的功能，使用户对系统的状态一目了然。

我们会基于上一章的 prod.yml，在本地运行这个组合。[3] 最理想的设置应当是将 ELK 容器放在另一个单独的主机上，以保持不同的功能之间清晰的划分。第 11 章将会讨论如何实现它；但为了简单起见，这一章我们会把所有东西都放在同一台主机上运行。

现在要做的第一件事，就是要先弄明白怎样把 Docker 的日志发送到 Logstash。为此，我们将使用一个名为 Logspout（https://github.com/gliderlabs/logspout）的 Docker 专用工具，

注 3：如果你想在云端运行它，那就去做吧，不过你可能会发现需要升级服务器才能运行所有服务。

它通过 Docker API 把日志以流的方式从运行容器传送给特定的端点（类似用于 Docker 的 rsyslog）。由于我们希望使用 Logstash 作为端点，我们还需要安装 logspout-logstash（https://github.com/looplab/logspout-logstash）这个适配器，它能够把 Docker 的日志转变为 Logstash 很容易就能读取的格式。Logspout 的设计尽可能小而高效，使它能够以最少的资源运行在每一台 Docker 主机上。为了实现这一目标，Logspout 的编程语言是 Go，而底层则用了体积极小的 Alpine Linux 镜像。由于 Logspout 的默认容器不包含 logstash 的适配器，我们将使用一个提前准备好的容器。

我们打算部署的整体设置如图 10-1 所示，左侧容器的日志由 logspout 整理，然后交由 Logstash 解析和过滤，接下来把结果放进 Elasticsearch。最后，Kibana 用于对 Elasticsearch 容器中的数据进行可视化及研究。

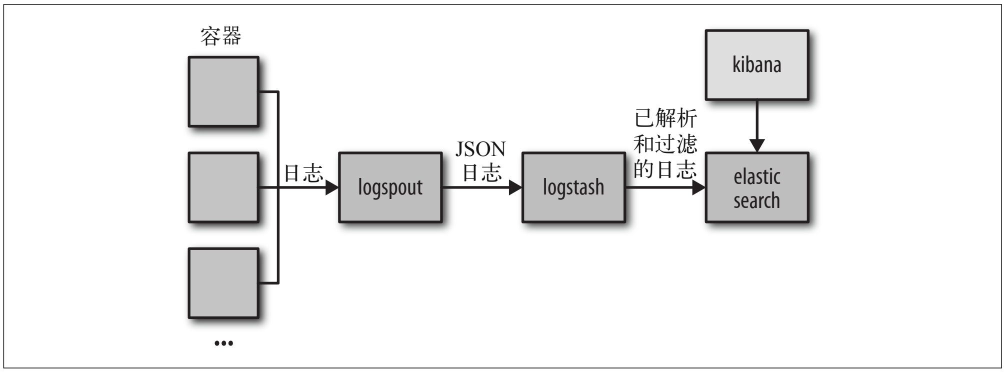


图 10-1：使用 Logspout 和 ELK 记录容器日志

首先创建一个包含以下内容的名为 prod-with-logging.yml 的新文件。

```
proxy:
  image: amouat/proxy:1.0
  links:
    - identidock
  ports:
   - "80:80"
  environment:
    - NGINX_HOST=45.55.251.164  ❶
    - NGINX_PROXY=http://identidock:9090
identidock:
  image: amouat/identidock:1.0  ❷
  links:
   - dnmonster
   - redis
  environment:
    ENV: PROD
dnmonster:
  image: amouat/dnmonster
redis:
  image: redis

logspout:
```

```
    image: amouat/logspout-logstash
    volumes:
      - /var/run/docker.sock:/tmp/docker.sock  ❸
    ports:
      - "8000:80"  ❹
```

❶ 把这里的 IP 地址换成你的主机的 IP 地址。
❷ 这里使用的是我在 Docker Hub 上的 identidock 镜像，你也可以换成你自己的版本。
❸ 挂载 Docker 的套接字，这样 Logspout 就可以连接到 Docker 的 API。
❹ 为了可以查看日志，我们把 Logspout 的 HTTP 接口公开。但在生产环境的系统中千万别把端口暴露出来。

现在，如果再次打开应用程序，应该能够连接到 Logspout 的 HTTP 串流接口：

```
$ docker-compose -f prod-with-logging.yml up -d
...
$ curl localhost:8000/logs
```

在浏览器中打开 identidock，你应该可以在终端上看到一些日志：

```
logging_proxy_1|192.168.99.1 - - [24/Sep/2015:11:36:53 +0000] "GET / HTTP/1....
logging_identidock_1|[pid: 6|app: 0|req: 1/1] 172.17.0.14 () {40 vars in 660...
logging_identidock_1|Cache miss
     logging_proxy_1|192.168.99.1 - - [24/Sep/2015:11:36:53 +0000] "GET /mon...
logging_identidock_1|[pid: 6|app: 0|req: 2/2] 172.17.0.14 () {42 vars in 788...
logging_identidock_1|[pid: 6|app: 0|req: 3/3] 172.17.0.14 () {42 vars in 649...
```

太棒了，这部分看来已经可以正常工作，将来你会发现这个界面非常有用。下一步需要将输出发送到有用的地方，在我们的例子中是 Logstash 容器。请注意，在一个多主机的系统中，你需要在每台主机上运行一个 Logspout 容器，它会负责把日志转发到一个中央的 Logstash 实例。现在就来把 Logstash 串联起来。你需要对 Compose 文件进行如下更新。

```
...
logspout:
  image: amouat/logspout-logstash
  volumes:
    - /var/run/docker.sock:/tmp/docker.sock
  ports:
    - "8000:80"
  links:
    - logstash  ❶
  command: logstash://logstash:5000  ❷
logstash:
  image: logstash
  volumes:
    - ./logstash.conf:/etc/logstash.conf  ❸
  environment:
    LOGSPOUT: ignore  ❹
  command: -f /etc/logstash.conf
```

❶ 添加一个到 Logstash 容器的连接。
❷ 使用“logstash”前缀告知 Logspout 在输出时使用 `Logstash` 模块。
❸ 把 Logstash 配置文件映射进来。

❹ Logspout 不会对任何在环境变量中设定了 `LOGSPOUT` 变量的容器收集日志。我们不希望从 Logstash 容器收集日志，因为这样做有导致循环的风险。如果有一个格式错误的日志信息导致 Logstash 产生错误，而这个错误被记录到日志中并被送回 Logstash，从而产生一个新的错误，这个过程就会循环不断。

配置文件应命名为 logstash.conf，并包含以下内容：

```
input {
  tcp {
    port => 5000
    codec => json  ❶
  }
  udp {
    port => 5000
    codec => json  ❶
  }
}

output {
  stdout { codec => rubydebug }  ❷
}
```

❶ 需要使用 `json` 编解码器来处理 Logspout 的输出。
❷ 为了测试，将日志输出至 `STDOUT`。

现在把程序运行起来，看看会发生什么：

```
$ docker-compose -f prod-with-logging.yml up -d
...
$ curl -s localhost > /dev/null
$ docker-compose -f prod-with-logging.yml logs logstash
...
logstash_1 | {
logstash_1 |            "message" => "2015/09/24 12:50:25 logstash: write u...
logstash_1 |        "docker.name" => "/logging_logspout_1",
logstash_1 |          "docker.id" => "d8f69d05123c43c9da7470951547b23ab32d4...
logstash_1 |       "docker.image" => "amouat/logspout-logstash",
logstash_1 |    "docker.hostname" => "d8f69d05123c",
logstash_1 |           "@version" => "1",
logstash_1 |         "@timestamp" => "2015-09-24T12:50:25.708Z",
logstash_1 |               "host" => "172.17.0.11"
logstash_1 | }
...
```

你应该会看到一些像前面一样的 Ruby 格式的代码。需要注意的是，输出中包含了一些如容器名称和它的 ID 之类的字段，它们都是由 Logspout 加进来的。Logstash 首先取得 JSON 格式输出，把它消化后，再把信息以 Ruby 调试的格式输出。但我们可以用到的 Logstash 功能还有很多，可以根据需求对日志进行过滤和更改。例如，你可能希望在日志传送到其他服务作进一步处理或存储之前，先把个人身份信息或敏感信息删除。而在我们的例子中，最好就是能够把 nginx 的日志信息拆分成其组成部分。要这样做的话，在 Logstash 配置文件中添加一个 `filter`（过滤器）就可以了：

```
input {
  tcp {
    port => 5000
    codec => json
  }
  udp {
    port => 5000
    codec => json
  }
}

filter {
  if [docker.image] =~ /^amouat\/proxy.*/ {
    mutate { replace => { type => "nginx" } }
    grok {
      match => { "message" => "%{COMBINEDAPACHELOG}" }
    }
  }
}

output {
  stdout { codec => rubydebug }
}
```

这个过滤器检查信息是否来自名为 amouat/proxy 的镜像。如果是的话，Logstash 在解析信息的时候，就会使用现有的 COMBINEDAPACHELOG 过滤器，在输出中还会增加一些额外的字段。当你添加了前面的过滤器并重新启动程序后，会发现日志信息变成了这样：

```
logstash_1 | {
logstash_1 |            "message" => "87.246.78.46 - - [24/Sep/2015:13:02:...
logstash_1 |        "docker.name" => "/logging_proxy_1",
logstash_1 |          "docker.id" => "5bffa4f4a9106e7381b22673569094be20e8...
logstash_1 |       "docker.image" => "amouat/proxy:1.0",
logstash_1 |    "docker.hostname" => "5bffa4f4a910",
logstash_1 |           "@version" => "1",
logstash_1 |         "@timestamp" => "2015-09-24T13:02:59.751Z",
logstash_1 |               "host" => "172.17.0.23",
logstash_1 |               "type" => "nginx",
logstash_1 |           "clientip" => "87.246.78.46",
logstash_1 |              "ident" => "-",
logstash_1 |               "auth" => "-",
logstash_1 |          "timestamp" => "24/Sep/2015:13:02:59 +0000",
logstash_1 |               "verb" => "GET",
logstash_1 |            "request" => "/",
logstash_1 |        "httpversion" => "1.1",
logstash_1 |           "response" => "200",
logstash_1 |              "bytes" => "266",
logstash_1 |           "referrer" => "\"-\"",
logstash_1 |              "agent" => "\"curl/7.37.1\""
logstash_1 | }
```

可以看到，过滤器能够把很多额外信息提取出来，如响应代码、请求类型和 URL。使用类似的技巧，你可以为各种镜像的日志记录设置不同的过滤器。

下一步就是将 Logstash 容器连接到 Elasticsearch 容器。同时，也会加上 Kibana 容器，它将为 Elasticsearch 提供界面。

把 Compose 文件更新如下。

```
...
logspout:
  image: amouat/logspout-logstash
  volumes:
    - /var/run/docker.sock:/tmp/docker.sock
  ports:
    - "8000:80"
  links:
    - logstash
  command: logstash://logstash:5000

logstash:
  image: logstash:1.5
  volumes:
    - ./logstash.conf:/etc/logstash.conf
  environment:
    LOGSPOUT: ignore
  links:
    - elasticsearch  ❶
  command: -f /etc/logstash.conf

elasticsearch:  ❷
  image: elasticsearch:1.7
  environment:
    LOGSPOUT: ignore

kibana:  ❸
  image: kibana:4.0
  environment:
    LOGSPOUT: ignore
    ELASTICSEARCH_URL: http://elasticsearch:9200
  links:
    - elasticsearch
  ports:
    - "5601:5601"
```

❶ 添加一个到 Elasticsearch 容器的连接。
❷ 基于 Elasticsearch 的官方镜像创建容器。
❸ 创建一个 Kibana 4.0 容器。注意，我们添加了一个到 Elasticsearch 容器的连接，并打开端口 5601 作界面访问。

我们还需要更新 logstash.conf 文件的输出部分，使输出导向我们的 Elasticsearch 容器：

```
...
output {
  elasticsearch { host => "elasticsearch" }  ❶
  stdout { codec => rubydebug }  ❷
}
```

❶ 把数据以 Elasticsearch 能够理解的格式输出到名为“elasticsearch”的远程主机。
❷ 你可以决定现在是否删除这一行；它仅仅是用于调试，而且它的信息是完全重复的。

现在重新启动程序，看看有什么变化：

```
$ docker-compose -f prod-with-logging.yml up -d
Recreating logging_dnmonster_1...
Recreating logging_redis_1...
Recreating logging_elasticsearch_1...
Recreating logging_kibana_1...
Recreating logging_identidock_1...
Recreating logging_proxy_1...
Recreating logging_logstash_1...
Recreating logging_logspout_1...
```

然后在浏览器中打开 `localhost`，随便对 identidock 进行一些操作，使我们的日志栈能有数据以供分析。准备就绪后，可以在浏览器中打开 `localhost:5601` 来访问 Kibana 程序。你应该会看到类似图 10-2 的画面。

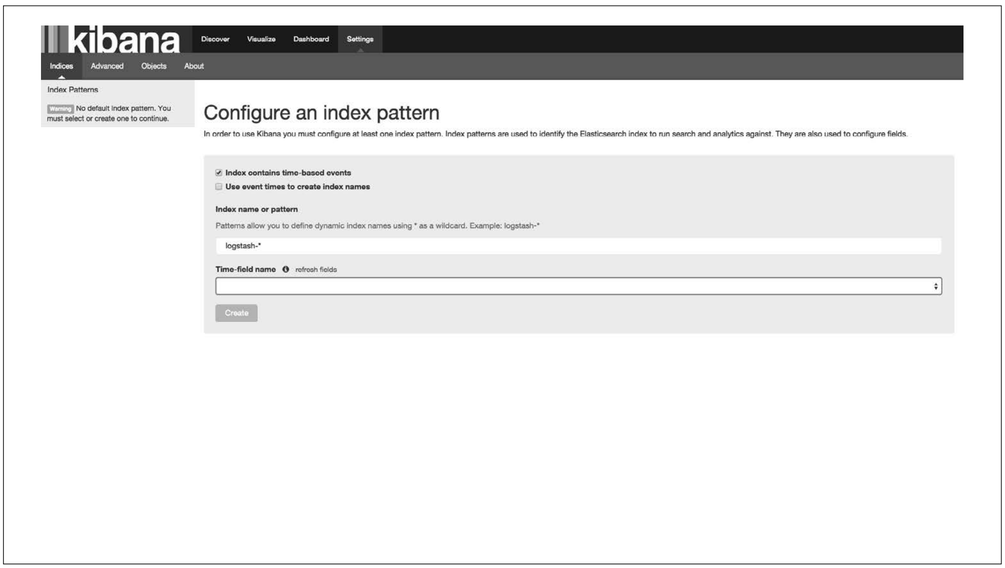


图 10-2：Kibana 配置页面

选择 `@timestamp` 作为 time-field name 的值，然后点击“Create”。你应该会看到一个 Elasticsearch 找到的所有字段的页面，其中包括 nginx 和 Docker 的。

如果现在点击“Discover”，应该会得到一个显示日志量柱状图的页面，柱状图的下面是一些最近的日志信息，如图 10-3 所示。

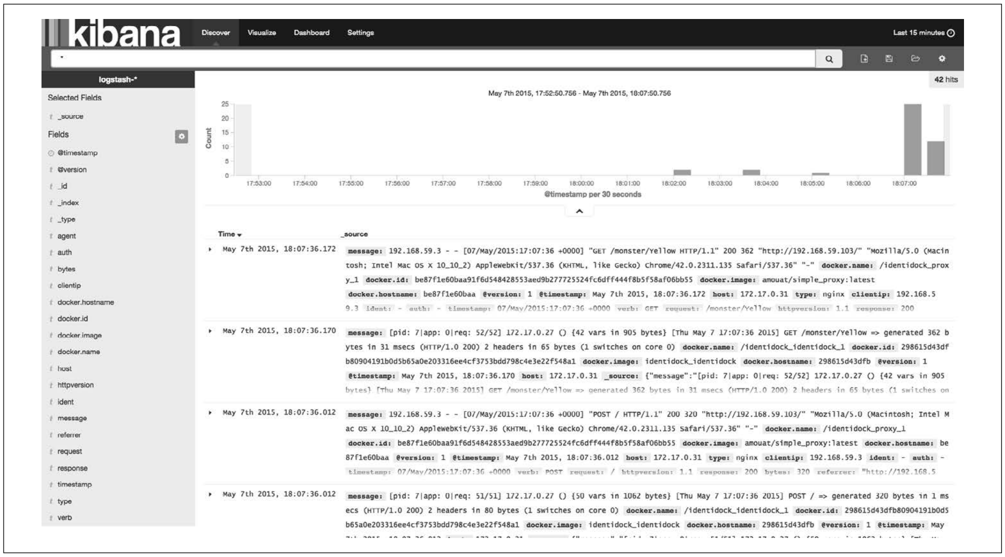


图 10-3：Kibana 的 Discover 页面

点击右上角的时钟图标就能轻松地更改时间段。日志可以通过搜索某些字段中出现的值进行过滤。例如，尝试在“message”字段中搜索“Cache miss”，可以得到在某时间段内缓存未命中的柱状图。更高级的图表和可视化分析可以通过“Visualize”标签页生成，包括线图、饼图，以及数据表格和自定义指标。

Kibana 3 和 4 的区别

如果你使用的 Kibana 是 4.0 以前的版本，你必须确保浏览器可以访问由 Elasticsearch 容器转发到主机的端口。这是由于 Kibana 是一个 JavaScript 应用，它是在客户端那边运行的。从版本 4 开始，连接是通过 Kibana 作代理，因此已经没有这个要求了。

这一章并不打算介绍高级日志分析的方方面面，因此 Kibana 的介绍就到此为止。总而言之，Kibana 和类似的解决方案提供了强大和高度可视化的方法来协助调查你的程序和数据。

日志存储和管理

无论你最终使用哪一个日志驱动和分析工具，你都需要决定如何保存日志以及保存多长时间。如果你从来没有想过这个问题，那么你的容器很可能正在使用默认的日志记录方式，它会慢慢地把所有硬盘空间用完，直到系统崩溃。

Linux 的 logrotate 工具可以用来管理日志文件的增长。通常情况下，日志文件会被分成不同的时间段保存下来，文件会在特定的时间间隔内按它们的先后顺序被迁移。举

个例子，除了当前的日志文件外，你可能还有一个旧版本的日志文件，甚至还有比这个更早一点的，以此类推。为了节省存储空间，早于当前的旧版本的日志和更早的日志都会被压缩。每天，当前的日志文件会被改为旧版本，而原来的旧版本的日志将会被压缩并改为更早一版的日志，以此类推，而原先最早的日志会被删除。

上述步骤用以下的 logrotate 配置就能做到，你可以把它作为新文件保存到 /etc/logrotate.d/ 目录下（例如 /etc/logrotate.d/docker），或者把内容添加到 /etc/logrotate.conf 中：

```
/var/lib/docker/containers/*/*.log {
  daily  ❶
  rotate 3  ❷
  compress  ❸
  delaycompress  ❸
  missingok  ❹
  copytruncate  ❺
}
```

❶ 每天执行日志迁移。
❷ 保存 3 个日志文件。
❸ 指示需要压缩日志，除了当前和最近的日志以外。
❹ 即使文件不存在，logrotate 也不会报错。
❺ 不会移动当前的日志文件，而是先复制它，然后把文件内容彻底删除（把大小设置为 0）。这是为了确保 Docker 不会因文件消失而不高兴。注意，在文件复制和删除内容之间，如果程序在这个时候写入日志，那么将有可能导致数据丢失。

你可能发现 logrotate 是默认由 cron 每天执行一次任务；如果你希望能更频繁地整理日志，那就需要修改 cron 任务。

如果你需要更持久和可靠的日志存储，可以把日志转发到一个可靠的数据库，如 PostgreSQL。在 Logstash 或同样的工具中，可以轻松地把它加上，作为另一个输出。不要单纯依靠像 Elasticsearch 这种索引方案作为存储，因为它们不具备和 PostgreSQL 这种发展成熟的数据库同等程度的容错保证。记住，如果有需要的话，可以使用 Logstash 过滤器来清除个人身份信息这类数据。

如何处理把日志记录到文件的应用

如果你的程序不是把日志写到 STDOUT 或 STDERR，而是记录到文件中的话，仍然有一些方法可以使用。如果你已使用 Docker 的 API 来处理日志（例如通过 Logspout 容器），那么最简单的办法就是运行一个进程（一般是 `tail -F`）负责把文件打印到 STDOUT。为了遵循一个容器只运行单一进程的哲学，比较漂亮的做法是利用另一个容器通过 `--volumes-from` 参数挂载日志文件。

例如，一个名叫“tolog”的容器声明 /var/log 作为它的数据卷，那么可以使用以下命令：

```
$ docker run -d --name tolog-logger \
             --volumes-from tolog \
             debian tail -F
/dev/log/*
```

如果你不喜欢这个方法，也可以将日志挂载到主机上的某个目录，然后对它们执行日志收集器，如 fluentd（http://www.fluentd.org/）。

10.1.4 通过syslog实现日志管理

假如你的 Docker 主机支持 syslog，那就可以使用 syslog 驱动，它能够将容器日志发送到主机上的 syslog 进程。用例子来说明就会很清楚。

```
$ ID=$(docker run -d --log-driver=syslog debian \
      sh -c 'i=0; while true; do i=$((i+1)); echo "docker $i"; sleep 1; done;')
$ docker logs $ID
"logs" command is supported only for "json-file" logging driver (got: syslog)  ❶
$ tail /var/log/syslog  ❷
Sep 24 10:17:45 reginald docker/181b6d654000[3594]: docker 48
Sep 24 10:17:46 reginald docker/181b6d654000[3594]: docker 49
Sep 24 10:17:47 reginald docker/181b6d654000[3594]: docker 50
Sep 24 10:17:48 reginald docker/181b6d654000[3594]: docker 51
Sep 24 10:17:49 reginald docker/181b6d654000[3594]: docker 52
Sep 24 10:17:50 reginald docker/181b6d654000[3594]: docker 53
Sep 24 10:17:51 reginald docker/181b6d654000[3594]: docker 54
Sep 24 10:17:52 reginald docker/181b6d654000[3594]: docker 55
Sep 24 10:17:53 reginald docker/181b6d654000[3594]: docker 56
Sep 24 10:17:54 reginald docker/181b6d654000[3594]: docker 57
```

❶ 在写作本书的时候，`docker logs` 命令仅适用于默认的日志记录方式。

❷ 在我的 Ubuntu 主机上，docker 的日志都被发送到 /var/log/syslog。其他 Linux 发行版的路径可能会不一样。

syslog 文件中虽然含有来自这个容器的消息，但它还会包含其他系统服务和其他容器的消息。由于日志消息里包含容器的 ID（以短格式记录），因此我们可以很容易地利用 `grep` 命令来查找有关某个容器的消息：

```
$ grep ${ID:0:12} /var/log/syslog  ❶
Sep 24 10:16:58 reginald docker/181b6d654000[3594]: docker 1
Sep 24 10:16:59 reginald docker/181b6d654000[3594]: docker 2
Sep 24 10:17:00 reginald docker/181b6d654000[3594]: docker 3
Sep 24 10:17:01 reginald docker/181b6d654000[3594]: docker 4
Sep 24 10:17:02 reginald docker/181b6d654000[3594]: docker 5
Sep 24 10:17:03 reginald docker/181b6d654000[3594]: docker 6
Sep 24 10:17:04 reginald docker/181b6d654000[3594]: docker 7
...
```

❶ 记录到日志的 ID 是短格式，因此只需使用完整 ID 的前 12 个字符。

Docker 事件 API

除了容器和 Docker 守护进程的日志外，还有另一组数据你可能也希望监控并对它作出反应，那就是 Docker 事件。在 Docker 容器的生命周期中，大部分阶段中发生的事件都会被记录下来。这些事件包括 `create`、`destroy`、`die`、`export`、`kill`、`pause`、`attach`、`restart`、`start`、`stop`，以及 `unpause`。它们当中大部分的意思应该都不言自明，但要注意 `die` 在容器退出时发生，而 `destroy` 则是在容器被删除时发生（即 `docker rm` 被调用时）。图 10-4 显示的是容器生命周期的图表。

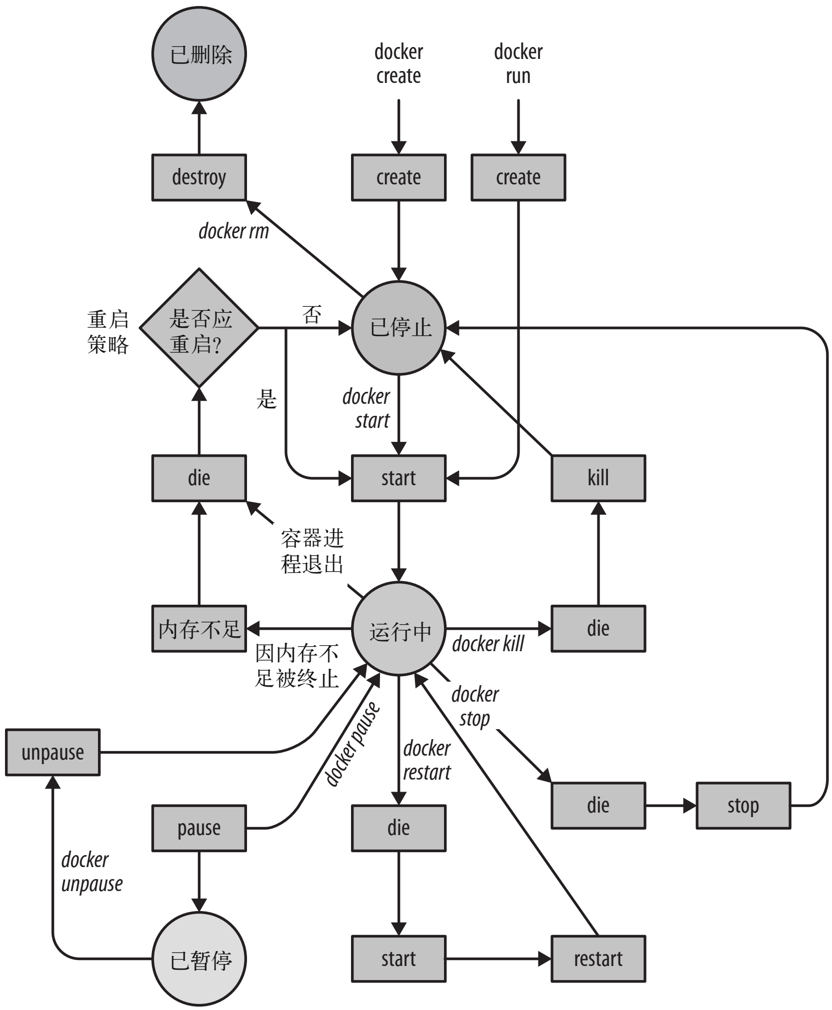


图 10-4：Docker 生命周期 [参照 Docker 官方文档（https://docs.docker.com/engine/reference/api/docker_remote_api/）绘制，原作者为 Gilder Labs 的 Matt Good，以 CC-BY-SA 4.0 许可证（https://creativecommons.org/licenses/by-sa/4.0/）发布]

关于镜像的事件有 untag 和 delete。当标签被删除时，即 docker rmi 命令成功执行后，untag 事件就会发生。当镜像被删除时，delete 事件便会发生（docker rmi 命令不一定会引发这个事件，因为镜像可能有多个标签）。时间戳会以 RFC 3339（https://www.ietf.org/rfc/rfc3339.txt）格式显示。

可以通过 docker events 命令来查看事件：

```
$ docker events
2015-09-24T15:23:28.000000000+01:00 44fe57bab...: (from debian) create
2015-09-24T15:23:28.000000000+01:00 44fe57bab...: (from debian) attach
2015-09-24T15:23:28.000000000+01:00 44fe57bab...: (from debian) start
2015-09-24T15:23:28.000000000+01:00 44fe57bab...: (from debian) die
```

由于 docker events 命令的输出是一个流，你需要先在另一个终端里执行一些 Docker 命令，然后才能够看到它输出一些结果。docker events 命令还接受一些参数，用于过滤结果和控制返回结果的时间段。可用于过滤的条件包括容器、镜像和事件。参数中使用的时间戳必须遵循 RFC 3339 的格式（例如“2006-01-02T15:04:05.000000000Z07:00”），或给出从 Unix epoch 开始过了多少秒（例如“1378216169”）。

当你希望对容器事件能够自动作出反应时，Docker 的事件 API 就能派上用场。例如，Logspout 程序通过 API 知道容器何时启动，从而开始转发容器日志。9.2 节的“强大的配置文件生成工具”中曾经提到过由 Jason Wilder 创作的 nginx-proxy，当容器启动时，它通过事件 API 来实现自动负载均衡。此外，你也可以简单地把数据记录下来，对容器的生命周期进行分析。

为了改善我们的方案，可以告诉 syslog 把 Docker 的日志记录在一个单独的文件中。Docker 会将日志写到 syslog 中称为“daemon”功能（facility）的类别中，这个 syslog 机制可以让你很轻松地把所有守护进程的消息写到同一个文件里，但要是把只属于 Docker 的消息单独筛选出来存储的话，就要费点工夫。[4] 如果你的 rsyslog 版本是 7 或以上（这个可能性很高），可以使用以下这个规则：

```
:syslogtag,startswith,"docker/" /var/log/containers.log
```

它会把所有 Docker 容器的消息全部放在 /var/log/containers.log。你需要把这个规则写到 rsyslog 的配置文件中。至少在 Ubuntu 上，可以创建一个新文件 /etc/rsyslog.d/30-docker.conf 并把规则写在里面。重启 syslog 后，日志便会出现在这个新文件中：

```
$ sudo service rsyslog restart
rsyslog stop/waiting
rsyslog start/running, process 15863
$ docker run -d --log-driver=syslog debian \
  sh -c 'i=0; while true; do i=$((i+1)); echo "docker $i"; sleep 1; done;'
$ cat /var/log/containers.log
Sep 24 10:30:46 reginald docker/1a1a57b885f3[3594]: docker 1
Sep 24 10:30:47 reginald docker/1a1a57b885f3[3594]: docker 2
```

注 4：本来应该很容易才对，但 syslog 有些部分似乎还停留在 20 世纪 80 年代……

```
Sep 24 10:30:48 reginald docker/1a1a57b885f3[3594]: docker 3
Sep 24 10:30:49 reginald docker/1a1a57b885f3[3594]: docker 4
Sep 24 10:30:50 reginald docker/1a1a57b885f3[3594]: docker 5
Sep 24 10:30:51 reginald docker/1a1a57b885f3[3594]: docker 6
Sep 24 10:30:52 reginald docker/1a1a57b885f3[3594]: docker 7
```

但现在相同的日志消息也会被保存到其他的日志文件中（例如 /var/log/syslog）。为了禁止这个行为，需要在我们的规则后面加上 `&stop`，像这样：

```
:syslogtag,startswith,"docker/" /var/log/containers.log
&stop
```

> **syslog 与 Docker Machine 虚拟机**
>
> 当我还在写这本书的时候，由 Docker Machine 所部署的 boot2docker 虚拟机并未默认运行 syslog。为了测试 syslog，你可以登录虚拟机并运行 syslogd。例如：
>
> ```
> $ docker-machine ssh default
> ...
> docker@default:~$ syslogd
> ```
>
> 要使这个改变永久化，可以修改 boot2docker 虚拟机中的 /var/lib/boot2docker/bootsync.sh 脚本文件来让它执行 syslogd，虚拟机会在启动 Docker 之前执行这个脚本。例如：
>
> ```
> $ docker-machine ssh default
> ...
> docker@default:~$ cat /var/lib/boot2docker/bootsync.sh
> #!/bin/sh
> syslogd
> ```
>
> 注意，boot2docker 虚拟机使用 busybox 默认的 syslog 实现，但它不如 rsyslogd 灵活。

在启动守护进程时加上 `--log-driver=syslog` 参数，便能把 syslog 设置为 Docker 容器的默认日志方式（一般是通过修改 Docker 服务的配置文件，譬如在 Ubuntu 上便可以把参数添加到 /etc/default/docker 中的 `DOCKER_OPTS`）。

使用rsyslog转发日志

我们还可以告诉 rsyslog 把日志转发到另一台服务器上，而不是存储在本地。这样做就可以为那些负责集中处理日志的服务提供日志，例如 Logstash 或另一台 syslog 服务器，好处是避免了使用如 Logspout 服务的额外开销。

要在我们的 identidock 范例中把 Logspout 替换成 rsyslog，需要修改 Logstash 的配置，让它准备接受来自 syslog 的输入，而且从主机上转发一个用于与 rsyslog 通信的端口到 Logstash，并告诉 rsyslog 通过网络传送日志，而不是保存到文件。

首先需要重新配置 Logstash，把它的配置文件进行如下修改。

```
input {
  syslog {
    type => syslog
      port => 5544
  }
}

filter {
  if [type] == "syslog" {
    syslog_pri { }
    date {
      match => [ "syslog_timestamp", "MMM d HH:mm:ss", "MMM dd HH:mm:ss" ]
    }
  }
}

output {
  elasticsearch { host => "elasticsearch" }
  stdout { codec => rubydebug }
}
```

现在需要修改 rsyslog 的配置。配置和之前几乎一模一样，只不过不再指定 /var/log/containers.log 文件，而是使用 `@@localhost:5544` 这样的语法。例如：

```
:syslogtag,startswith,"docker/" @@localhost:5544
&stop
```

它的意思是告诉 rsyslog 把日志通过 TCP 发送到本地的 5544 端口。如果要使用 UDP，只需一个 `@` 即可。[5]

配置的最后一步便是改写 Compose 文件。在改写之前，最好先停止任何正在运行的 identidock 实例：

```
$ docker-compose -f prod-with-logging.yml stop
...
```

现在可以放心地把 Logspout 从 Compose 文件中移除，并且在主机上发布一个能让 rsyslog 与 Logstash 通信的端口：

```
...
logstash:
  image: logstash:1.5
  volumes:
    - ./logstash.conf:/etc/logstash.conf
  environment:
    LOGSPOUT: ignore
  links:
    - elasticsearch
    - "127.0.0.1:5544:5544"   ❶
  command: -f /etc/logstash.conf

elasticsearch:
```

注 5：这个语法够隐晦的吧！

```
    image: elasticsearch:1.7
    environment:
      LOGSPOUT: ignore

  kibana:
    image: kibana:4.1
    environment:
      LOGSPOUT: ignore
      ELASTICSEARCH_URL: http://elasticsearch:9200
    links:
      - elasticsearch
    ports:
      - "5601:5601"
```

❶ 发布 5544 端口。端口只绑定到 `127.0.0.1` 之上，因此只允许本机与端口连接，而网络上的其他机器则不能。

最后一步就是重启 rsyslog 和 identidock。现在可以看到，日志已经通过 rsyslog 发送到 Logstash，而不是使用速度较慢的 Logspout 方法。虽然我们还要配置过滤器，才能获得与 Logspout 之前取得的一样的信息，并将其转发到 Logstash，不过利用 rsyslog 转发日志却是一个非常有效和可靠的解决方案。

受保障的日志记录

无论你是否意识到这一点，当设计你的日志记录架构时，你是在效率和百分之百的准确性及可靠性之间作取舍。如果你的日志只是用于调试和监控，那么只需选择一个你认为最简单的方案就可以了。但是，如果有一些日志信息在发生之后必须立即发出警报，或者为了遵循核查的政策而必须保证日志完整无缺，那么一定要考虑清楚你的日志架构中各个连接部分的性质及保障程度。

下面是一些你需要考虑的重点。

- 传送日志时使用什么传输协议？虽然 UDP 速度较快，但就可靠程度而言，它能够提供的保障的可靠度比 TCP 要低（不过 TCP 也不能保证是可靠的）。
- 网络中断后将会发生什么事情？很多工具（包括 rsyslog）都有在远程服务器能够重新连接之前先暂时缓冲信息的配置选项。
- 如何存储和备份日志信息？与文件系统相比，数据库能够提供更高的可靠性和更有力的容错保障。

在考虑以上各点的时候，还需要一并考虑日志的安全性；你的日志很可能包含敏感信息，因此访问权的控制非常重要。任何在公共的互联网上传递的日志必须经过加密，并且只有合适的人群才可以访问已保存的日志。

10.1.5　从文件抓取日志

转发日志的另一种有效方法就是访问文件系统中的原始日志。如果你使用的是默认的日志记录方式，目前 Docker 把容器的日志文件保存在 /var/lib/docker/container/<container id>/<container id>-json.log。

直接从文件获取日志是个很有效率的做法，但缺点是它依赖于 Docker 内部如何实现，而并非透过公开的 API。出于这个原因，当 Docker 引擎更新后，基于这个方法的日志方案很可能再也不能工作。

10.2 监控和警报

在微服务系统中，你可能有几十、几百甚至几千个正在运行的容器。因此，为了监控运行中的容器和系统整体的状态，你能得到的帮助越多越好。一个优秀的监控方案应该能够一目了然地显示系统的运行状况，并在资源不足时提前发出警告（如磁盘空间、CPU 和内存）。我们也希望在发生任何状况的时候能够收到警报（例如，如果处理请求所需的时间变为几秒或更长）。

10.2.1 使用Docker工具进行监测

Docker 自带一个基本的命令行工具，名为 `docker stats`，能够返回一个资源使用情况的实时流。这个命令接受一个或多个容器名称作为参数，并打印各种关于这些容器的统计数据，与 Unix 程序 `top` 很类似。例如：

```
$ docker stats logging_logspout_1
CONTAINER            CPU %  MEM USAGE/LIMIT    MEM %  NET I/O
logging_logspout_1   0.13%  1.696 MB/2.099 GB  0.08%  4.06 kB/9.479 kB
```

统计数据包含 CPU 和内存的使用率，以及网络的利用率。需要注意的是，除非你对容器设置了内存限制，否则你看到的内存限制是主机的内存总量，而不是容器的可用内存量。

获取所有运行中容器的统计数据

在大部分时候，你会希望能够获取主机上所有运行中容器的统计数据（我认为这应该是个默认行为）。要做到这一点，只需要一点 shell 脚本技巧就可以了。

```
$ docker stats \
   $(docker inspect -f {{.Name}} $(docker ps -q))
CONTAINER                  CPU %  MEM USAGE/LIMIT  ...
/logging_dnmonster_1       0.00%  57.51 MB/2.099 GB
/logging_elasticsearch_1   0.60%  337.8 MB/2.099 GB
/logging_identidock_1      0.01%  29.03 MB/2.099 GB
/logging_kibana_1          0.00%  61.61 MB/2.099 GB
/logging_logspout_1        0.14%  1.7 MB/2.099 GB
/logging_logstash_1        0.57%  263.8 MB/2.099 GB
/logging_proxy_1           0.00%  1.438 MB/2.099 GB
/logging_redis_1           0.14%  7.283 MB/2.099 GB
```

首先 `docker ps -q` 命令用来获取所有运行中容器的 ID，以此作为 `docker inspect -f {{.Name}}` 的输入，它会把 ID 转换成名称，继而用作 `docker stats` 的输入。

若使用得当，这个命令会很有用，同时还意味着 Docker 可能已有一个能够给程序获取这方面统计数据的 API。这个 API 确实存在，你可以调用 /containers/<id>/stats 这个端点来获得容器上各种统计的输出流，它比命令行得到的结果更详细。但是这个 API 却不太灵

活；它的输出要么是每秒更新一次，要么只能输出一次后便结束，并没有选项来控制输出的频率或对输出进行过滤。这意味着通过这个统计数据的 API 进行连续的监控，对你来讲开销太大；尽管如此，它对一些临时的查询和调查仍然有用。

大部分 Docker 发布的各种指标也可以通过 Linux 内核的 CGroups 及命名空间直接获取，有各种不同的程序库和工具能够提供这个功能，包括 Docker 的 runc 库（https://github.com/opencontainers/runc）在内。如果需要监控某个特定数据，可以通过 runc 或直接调用内核函数来实现一个高效的方案。你需要使用一个允许调用底层内核函数的编程语言，例如 Go 或 C。实现的时候有一些陷阱必须注意，譬如如何避免更新数据时 fork 出新的进程。Docker 网站上的“Runtime Metrics”（https://docs.docker.com/engine/admin/runmetrics/）这篇文章对如何实现有所解说，并且深入讲解了有哪些指标数据能够从内核获取。一旦得到所需的数据，你应该会对以下工具感兴趣：用于汇总及计算指标的 statsd（https://github.com/etsy/statsd）；用于存储的 InfluxDB（http://influxdb.com/）及 OpenTSDB（http://opentsdb.net）；用于展示数据的 Graphite（http://graphite.readthedocs.org）及 Grafana（https://github.com/grafana/grafana）。

利用 Logstash 进行监控和发布警报

虽然 Logstash 原本是一个日志记录工具，但值得注意的是，Logstash 还能够提供一定程度的监听功能，而日志本身是一个很重要的监控对象。

例如，你可以检查 nginx 中的状态码，当发现接收到大量 500 状态码时，就自动以电子邮件或短信方式发出警报。Logstash 对很多常用的监控方案都备有合适的输出模块，其中包括 Nagios、Ganglia 和 Graphite。

大多数情况下，你应使用现有的工具来收集和汇总数据，并以可视化方式展示结果。关于这方面的功能，市面上已经有很多商用的解决方案，但本书将探讨主流的开源方案以及专门为容器创造的解决方案。

10.2.2　cAdvisor

谷歌的 cAdvisor（Container Advisor 的缩写）是最常用的 Docker 监控工具。它对主机上运行的容器以图形化的方式展示资源使用情况及性能指标的总览。

由于 cAdvisor 本身以容器提供，我们瞬间就能够把它运行起来。按照以下的命令和参数就能启动 cAdvisor：

```
$ docker run -d \
  --name cadvisor \
  -v /:/rootfs:ro \
  -v /var/run:/var/run:rw \
  -v /sys:/sys:ro \
  -v /var/lib/docker/:/var/lib/docker:ro \
  -p 8080:8080 \
  google/cadvisor:latest
```

如果你的主机使用红帽（或 CentOS），你还需要加上 `--volume=/cgroup:/cgroup` 参数来挂载 cgroup 的文件夹。

当容器启动后，在浏览器中打开 http://localhost:8080。你应该看到一个类似图 10-5 的网页，其中有很多不同的图表。点击“Docker Containers”链接，然后点击感兴趣的容器名称，便能够查看该容器的详细信息。

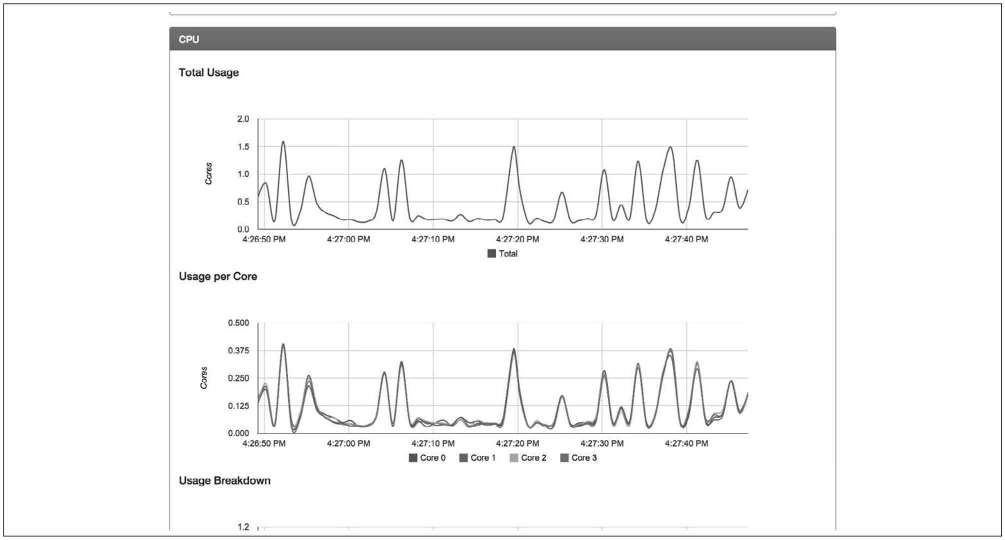


图 10-5：cAdvisor 中关于 CPU 使用率的图表

cAdvisor 能够汇总和处理各种统计数据，其他人可以通过 REST API 获取，方便进一步的处理和存储。InfluxDB 是专为存储和查询时间序列数据而设计的数据库，例如指标或统计分析等数据。在 cAdvisor 的路线图中提到一些功能，包括如何提升和调整容器性能，以及为集群编排和调度工具提供使用情况的预测信息。

10.2.3 集群解决方案

cAdvisor 很不错，但它只能针对一台主机工作。如果你运行的是一个庞大的系统，那么你希望能够得到的统计数据不只是每台主机本身的，还应该包括横跨所有主机的数据。有时候你想了解关于一组容器运行状况的数据，该组容器可能代表一些子系统，或多个容器实例中的某个功能。例如，你可能想查看所有 nginx 容器的内存使用量，或者一组正在运行数据分析的容器的 CPU 使用率。由于这种指标往往与特定的应用程序和问题相关，一套好的解决方案应具备一个可用于构建新指标和图表的查询语言。

谷歌已经开发出一套建立在 cAdvisor 之上的集群监控解决方案，称为 Heapster。但在本书编写的时候，它只能支持 Kubernetes 和 CoreOS，因此不会对它多作讨论。

接下来准备介绍的是 Prometheus，它是由 SoundCloud 开发的一个开源集群监控解决方案，能够接受众多不同的来源作为输入，其中包括 cAdvisor。它的目的是要支持大型的微服务架构系统，目前已在 SoundCloud 和 Docker 公司正式使用。

Prometheus

Prometheus（http://prometheus.io）的工作原理比较特别，因为它在基于*主动读取*（pull）的

模式上运行。它假设应用程序本身开放了用于获取统计数据的接口，然后由 Prometheus 服务器读取，而不是把数据直接发送给 Prometheus。Prometheus 的界面可用于查询数据，并能够以交互方式绘制图表，而另外提供的 PromDash 可用于将图形、图表和量表保存到仪表盘。Prometheus 还有一个称为 Alertmanager 的组件，能够汇集和拦截警报，并把它们转发到通知服务，可以是电子邮件，也可以是专门的服务，如 PagerDuty（http://www.pagerduty.com）和 Pushover（https://pushover.net）。

下面来看看怎样把 Prometheus 集成到 identidock。虽然我们不打算添加任何特殊指标，但只需在代码中使用它所提供的 Python 客户端程序库就能轻松做到。

接下来要做的是把 Prometheus 连接到我们的 cAdvisor 容器。我们也可以利用 container-exporter（https://github.com/docker-infra/container_exporter）项目，它采用的也是 Docker 的 libcontainer 库。如果你的 cAdvisor 容器还在运行，那么可以在 `/metrics` 接口看到它开放给 Prometheus 的指标：

```
$ curl localhost:8080/metrics
# HELP container_cpu_system_seconds_total Cumulative system cpu time consume...
# TYPE container_cpu_system_seconds_total counter
container_cpu_system_seconds_total{id="/",name="/"} 97.89
container_cpu_system_seconds_total{id="/docker",name="/docker"} 40.66
container_cpu_system_seconds_total{id="/docker/071c24557187c14fb7a2504612d4c...
container_cpu_system_seconds_total{id="/docker/1a1a57b885f33d2e16e85cee7f138...
...
```

启动 Prometheus 容器非常简单，但必须先创建一个配置文件。首先把以下内容保存到 prometheus.conf 文件：

```
global:
  scrape_interval: 1m  ❶
  scrape_timeout: 10s
  evaluation_interval: 1m

scrape_configs:

- job_name: prometheus
  scheme: http  ❷
  target_groups:
  - targets:
    - 'cadvisor:8080'
    - 'localhost:9090'  ❸
```

❶ 告诉 Prometheus 每隔 5 秒获取统计数据一次，5 秒可以说是一个比较高的数值。在生产环境中，抓取数据是一个成本，旧数据则是另一个你要承担的成本，你所选取的时间间隔必须在这两个成本之间取得平衡。
❷ 告诉 Prometheus 抓取 cAdvisor 数据的 URL（我们将使用 Docker 连接来设置主机名）。
❸ 在 9090 端口抓取 Prometheus 本身的指标数据。

用下面的命令启动 Prometheus 容器：

```
$ docker run -d --name prometheus -p 9090:9090 \
             -v $(pwd)/prometheus.conf:/prometheus.conf \
```

```
    --link cadvisor:cadvisor \
    prom/prometheus -config.file=/prometheus.conf
```

现在你应该能够在 http://localhost:9090 打开 Prometheus。主页会显示一些 Prometheus 的配置信息，以及它正在抓取的数据的端点状态。在“Graph”标签页中，你可以对 Prometheus 的数据进行研究。Prometheus 有自己的查询语言，并支持过滤器、正则表达式和各种运算符。举个简单的例子，尝试输入表达式：

```
sum(container_cpu_usage_seconds_total {name=~"logging*"}) by (name)
```

通过它，我们可以得到每个 identidock 容器在不同时间的 CPU 使用率。其中的 `{name=~"logging*"}` 表达式过滤不在我们的 Compose 应用程序中的容器（例如 cAdvisor 和 Prometheus 本身）。你需要把“logging”替换为你的 Compose 项目名称或文件夹名称。表达式中用到了 `sum` 函数，因为 CPU 使用率是以每个 CPU 为单位的。你得到的结果应类似图 10-6 所示。



图 10-6：用 Prometheus 绘制的容器 CPU 使用率图表

我们还可以更进一步，利用 PromDash 容器建立一个仪表盘。由于配置相对简单，我把这部分留给读者作为练习。图 10-7 展示的是一个仪表盘，它显示了上述的 CPU 指标和内存使用情况的图表。PromDash 还支持显示来自多个分散的 Prometheus 实例的图表，对需要显示不同地理位置或不同部门的图表非常有用。

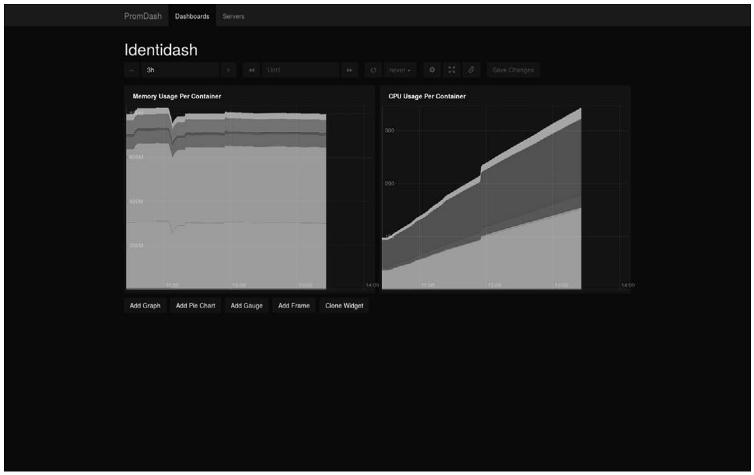


图 10-7：identidock 容器的 PromDash 仪表盘

当然，这只是很基本的 Prometheus 用法。实际的使用场景将涉及从分布在不同地方的主机抓取数据，而且端点的数量会多得多，并需要利用 PromDash 设置仪表盘和对可视化图表进行深入分析，以及利用 Alertmanager 发送通知。

10.3　商用的监听及日志记录解决方案

在本章中我刻意地只对开源和非托管形式的解决方案作了深入探讨，但其实现在已经有很多较为成熟的商用解决方案，而且竞争非常激烈。由于这个领域的发展迅速，与其告诉你有哪些具体的解决方案，不如说这个领域很值得你去花点时间仔细研究一下，尤其是当你需要寻找一个成熟的，或能提供托管服务的解决方案时。

10.4　总结

有效的日志记录和监听，对于运行基于微服务架构的应用而言至关重要。本章介绍了如何使用 ELK 组合以及 cAdvisor 和 Prometheus，从而让 identidock 能实现有效的日志记录和监听。虽然这个解决方案相较于我们的应用程序本身更笨重（将日志记录和监听本身与指标数据相比），但我们也看到，建立一个有效的方案是非常快速和简单的。

未来我们将会看到 Docker 对日志记录提供更多的支持和选项。在日志记录、监听及警报通知这几方面，都已经有很强大的商用产品可供选择，我们期待这些厂商未来能发展出专门为 Docker 和微服务设计的方案。

第三部分

工具和技术

第三部分将深入讲解安全可靠地运行 Docker 容器集群时所需要的工具和技术。

这一部分首先会介绍联网和服务发现。一旦运行容器的主机数目超过一台，服务发现就变得很重要。换言之，你的容器怎样才能找到彼此？你又该怎样把它们连接起来？

接下来会介绍能够帮助你实现服务编排和容器集群的解决方案。这些工具可以帮助开发者解决负载均衡、扩展以及故障切换等问题，并且可以帮助运营部门调度容器以及将资源利用最大化。任何长时间运行的应用程序迟早都会遇到这些问题，因此预先了解这些问题以及解决它们的可行方案将非常有用。

最后一章将介绍如何确保容器和微服务部署的安全性。容器给安全性带来了新的挑战，但也衍生出新的工具和技术。尽管这个主题放在了本书的最后，但它其实非常重要，只要你的工作涉及容器，你就应该深入了解它。

第 11 章

联网和服务发现

在容器的世界里，服务发现和联网之间的界线可以是非常模糊的。服务发现是为某个服务的客户端[1]自动提供连接至该服务的合适[2]实例的信息（通常是 IP 地址和端口）的过程。这个问题在静态的、单主机系统里很容易解决，因为总共只有一个实例，但在分布式系统里就复杂多了，因为一个服务会有多个实例，而且实例不会一直存在，它们都是动态创建和关闭的。服务发现的一种实现方法是让客户端只用服务名称发出请求（例如 `db` 或 `api`），然后在后台以特殊方法把名称对应到合适的地址。其中的“特殊方法”可以是以下几种：简单的大使容器、服务发现方案（譬如 Consul）、联网方案（譬如 Weave，它还包括服务发现功能），或者是以上几种的组合。

关于联网，我们关注的是如何把容器连接起来的过程。它不涉及物理以太网线的连接，尽管它常常与软件上对等的东西（如 veth）相关。容器联网的前提是假设主机之间已有可通信的路由，无论路由是穿越公网，还是只涉及本地高速交换机的内网。

因此，服务发现允许客户端发现服务实例，而联网则负责把相关部分连接起来。关于联网和服务发现的方案往往在功能上有所重叠，因为服务发现可以跨越不同网络，而联网的解决方案通常包含一些服务发现功能（例如 Weave）。像 Consul 这种提供纯服务发现的方案，很可能在健康检查、故障切换和负载均衡等方面提供了更丰富的功能。专门的联网解决方案则提供多种容器连接与路由[3]方法，还提供了传输加密和隔离容器组等功能。

注 1：这里的“客户端”是广义上的意思。这里所指的客户端不单是应用程序和在后台运行的服务，同时还包括对等体（试想在一个集群中，可能会包含互相协作的并行实例）和最终用户的客户端，比如浏览器。

注 2：什么实例“合适”取决于使用场景，它的意思可能是“任何一个”“速度最快的”或“最接近我的数据的”，等等。

注 3：即使“只是”使用服务发现，也将涉及部分联网功能：可能是使用默认的 Docker“网桥”联网模式时，把端口开放给主机；或者使用“主机”联网模式。二者都会涉及端口管理。

很多时候，你需要同时使用服务发现和联网两种方案（某种程度的联网是必需的）。具体需要怎样的方案则取决于你的实际情况，目前最佳实践仍在不断发展中。联网的实现可能在开发、测试和生产环境中有所不同，但服务发现通常需要考虑的问题是在应用层，因此并不会因环境不同而有所改变。

本章尝试由浅入深地讲解联网和服务发现的各个方面，从最简单的跨主机方案“大使容器”开始，之后会讲到诸如 etcd 和 Consul 的服务发现方案，最后才会详述 Docker 联网的细节以及相关的解决方案，例如 Weave、Flannel 和 Project Calico。

11.1 大使容器

将不同主机上的容器连接起来的一种方法叫作*大使容器*（ambassador）。它其实是代理容器，代替真正的容器（或服务）接收网络通信，并把流量转发至真正的服务中去。大使容器还能提供关注点分离（separation of concern），这使它不仅能用于连接不同主机的服务，还能在许多别的情况下发挥所长。

大使容器的主要优点在于，它允许在无需修改任何代码的情况下，让生产环境的联网架构有别于开发环境。开发人员可以放心使用本地的数据库和其他资源，因为他们知道运维人员在不需要修改任何代码的情况下，就能够把应用程序改为使用集群服务或远程资源。大使容器还可以随时切换它正在使用的支撑服务，但如果是使用 Docker 连接直接绑定到那个服务的话，客户端那边的容器就必须重新启动。

大使容器的缺点是，它需要更多的配置工作以及额外的开销，而且它是个潜在的故障点。当需要多个连接的时候，它很快就会变得非常复杂，造成管理上的一个负担。

图 11-1 展示了一个典型的开发场景，其中开发人员通过 Docker 连接把应用程序直接连接到数据库容器，而他的应用程序和数据库容器都是运行在他的本地笔记本电脑上。这个配置方便在开发过程中把东西推倒重来，非常适用于需要进行快速修改的开发流程。

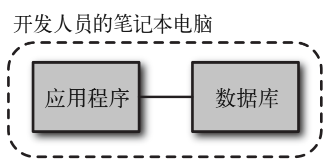


图 11-1：未使用大使容器的开发环境

图 11-2 中展示了一个普遍用于生产环境的设置，运维人员利用大使容器将应用程序与它所需的正式服务连接上，而这个服务运行在另一台服务器上。运维人员需要做的是配置大使容器，使网络通信能够通过它传给该服务，并利用 Docker 连接把应用程序与大使容器连上。无论是这个新设置，还是之前未使用大使容器的设置，代码中使用的主机名和端口都是一样的。

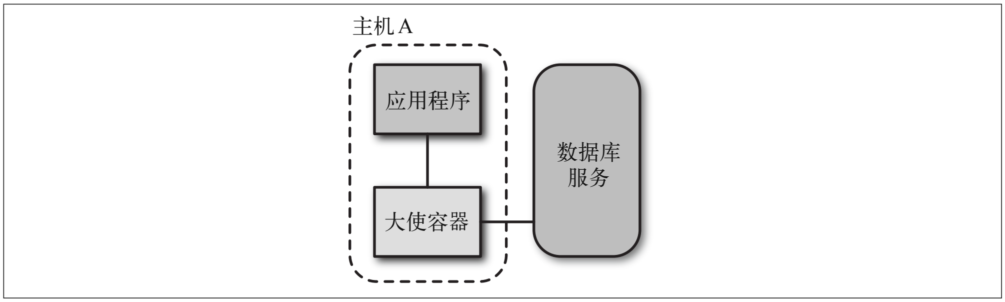


图 11-2：程序在生产环境中通过大使容器连接至所需服务

在图 11-3 所示的设置中，可以看到一个应用程序正与一个位于远程主机上的容器通信，而在这个远端的容器前面放置了一个大使容器。这种设置让我们只需更改大使容器，就能让远程主机把网络通信转发至位于其他地址上的新容器。同样，这个设置并不涉及任何代码的修改。

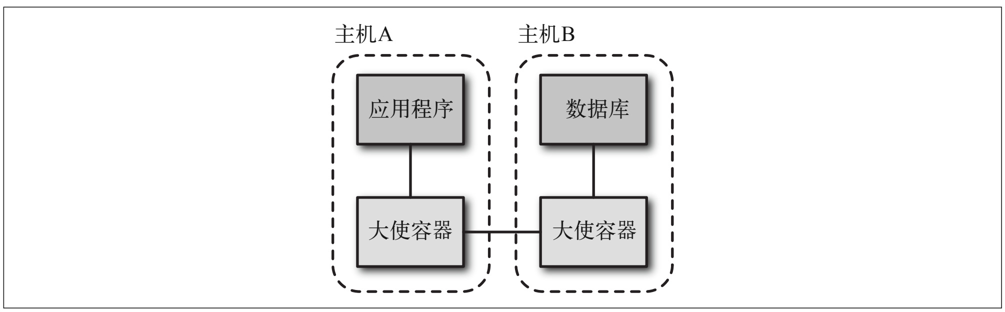


图 11-3：利用两个大使容器连接至位于远端的容器

大使容器本身可以是一个非常简单的容器。它需要做的只是建立应用程序与服务之间的连接。大使容器并没有官方镜像，你需要自己创建一个，或从 Docker Hub 上挑选一个用户镜像使用。

amouat/ambassador 镜像

这一章使用一个称为 amouat/ambassador 的镜像。这个镜像移植自 Sven Dowideit 的大使镜像（https://hub.docker.com/r/svendowideit/ambassador/），它经过了一些修改，包括改用了 alpine 基础镜像，以及在 Docker Hub 上配置了自动生成镜像。

这个镜像使用 socat（http://www.dest-unreach.org/socat/）来设置大使容器和目标之间的流量转发。它通过使用与 Docker 连接产生的环境变量相同的形式（例如 REDIS_PORT_6379_TCP=tcp://172.17.0.1:6379）来定义转发的目标位置。这意味着如果转发到本地连接的容器（如图 11-3 中的主机 B），只需少量配置即可。

由于该镜像以极简的 Alpine Linux 发行版为基础，它的大小仅 7MB 多一点，因此下载时间很短，而且几乎不会给系统增加任何负担。

下面来看看如何通过大使容器，像图 11-3 那样把我们的 identidock 应用与运行在另一台主机上的 Redis 容器相连接。我们将利用 Docker Machine 部署两个 VirtualBox 虚拟机（详见 9.1 节），但使用云端的 Docker 主机也能很轻松地执行这个范例。先准备好主机：

```
$ docker-machine create -d virtualbox redis-host
...
$ docker-machine create -d virtualbox identidock-host
...
```

现在需要在 redis-host 上建立一个 Redis 容器（称为 real-redis）和一个大使容器（称为 real-redis-ambassador）：

```
$ eval $(docker-machine env redis-host)
$ docker run -d --name real-redis redis:3
Unable to find image 'redis:3' locally
3: Pulling from redis
...
60bb8d255b950b1b34443c04b6a9e5feec5047709e4e44e58a43285123e5c26b
$ docker run -d --name real-redis-ambassador \
             -p 6379:6379 \  ❶
             --link real-redis:real-redis \  ❷
             amouat/ambassador
be613f5d1b49173b6b78b889290fd1d39dbb0fda4fbd74ee0ed26ab95ed7832c
```

❶ 发布主机上用于远程连接的 6379 端口。

❷ 大使容器通过使用连接中的 real-redis 容器的环境变量，把来自 6379 端口的请求转发到 real-redis 容器。

现在需要在 identidock-host 上建立一个大使容器：

```
$ eval $(docker-machine env identidock-host)
$ docker run -d --name redis_ambassador --expose 6379 \
         -e REDIS_PORT_6379_TCP=tcp://$(docker-machine ip redis-host):6379 \  ❶
         amouat/ambassador
Unable to find image 'amouat/ambassador:latest' locally
latest: Pulling from amouat/ambassador
31f630c65071: Pull complete
cb9fe39636e8: Pull complete
3931d220729b: Pull complete
154bc6b29ef7: Already exists
Digest: sha256:647c29203b9c9aba8e304fabfd194429a4138cfd3d306d2becc1a10e646fcc23
Status: Downloaded newer image for amouat/ambassador:latest
26d74433d44f5b63c173ea7d1cfebd6428b7227272bd52252f2820cdd513f164
```

❶ 我们需要手动设置一个环境变量，让大使容器知道如何连接远程主机。远程主机的 IP 地址可以通过 docker-machine ip 命令获得。

最后，把 identidock 连接到大使容器，然后启动 identidock 和 dnmonster：

```
$ docker run -d --name dnmonster amouat/dnmonster:1.0
Unable to find image 'amouat/dnmonster:1.0' locally
1.0: Pulling from amouat/dnmonster
...
c7619143087f6d80b103a0b26e4034bc173c64b5fd0448ab704206b4ccd63fa
```

```
$ docker run -d --link dnmonster:dnmonster \
             --link redis_ambassador:redis \
             -p 80:9090 \
              amouat/identidock:1.0
Unable to find image 'amouat/identidock:1.0' locally
1.0: Pulling from amouat/identidock
...
5e53476ee3c0c982754f9e2c42c82681aa567cdfb0b55b48ebc7eea2d586eeac
```

来试一试吧：

```
$ curl $(docker-machine ip identidock-host)
<html><head>...
```

成功了！我们没有修改任何代码，只使用了两个很小的大使容器，就把 identidock 分布在两台主机上运行了。虽然这样做看起来有点繁琐，而且还需要使用额外的容器，但它的设计其实是很简单和灵活的。我们很容易就能想出大使容器更复杂的使用场景，略举如下。

- 在不可信的网络连线上进行通信加密。
- 当容器开始监听 Docker 的事件流时，自动连接容器。
- 用作读写数据的代理，把数据读取和写入的请求分流至不同的服务器，读取请求使用的是只读的数据库。

在任何情况下，客户端都无需知道大使容器在做什么。

虽然大使容器很有用，但如果希望实现查找和连接远程的服务和容器，在大多数情况下使用联网和服务发现会更容易，而且可扩展性会更好。

11.2 服务发现

在本章的开始，服务发现定义为：*为某个服务的客户端自动提供连接至该服务的合适实例信息的过程*。

对于客户端的应用程序，这意味着它需要以某种方式请求或被告知服务的地址。接下来会介绍要求客户端明确调用 API 以请求服务地址的方法，以及可以轻松与现有应用程序集成的基于 DNS 的解决方案。

本节涵盖了现今能够与 Docker 搭配的主流服务发现方案。首先将深入剖析 etcd、Consul 和 SkyDNS，之后会对其他一些值得一提的方案作一个简短的介绍。它们原本是为大型分布式系统而设计的，并非专门针对容器而造。

11.2.1 etcd

etcd 是一个分布式的键值存储。它使用 Go 语言实现了 Raft Consensus 算法（https://raft.github.io/），以高效和容错为设计目标。Consensus（*共识*）是多个成员对数据达成一致意见的过程。当遇到故障和错误的时候，这个过程很快就会变得相当复杂。这个算法能保证数据的一致性，并且只需大多数成员在场，便能够添加新的值。

在一个 etcd 的集群中，每个成员都需要运行一个 etcd 二进制程序的实例，这个实例将与其

他成员进行通信。客户端访问 etcd 的方法是通过利用所有成员都配备的 REST 接口实现的。

etcd 集群的建议大小至少为 3 个成员，这样才能在故障发生时实现容错。不过下面的例子里只有两个成员，因为我们的目的只是为了展示 etcd 如何工作。

最佳集群大小

etcd 和 Consul 建议的集群大小为 3、5 或 7，这样能够在容错能力与性能之间取得平衡。

如果集群中只有一个成员，当故障发生时，数据就会丢失。如果只有两个成员，而其中一个发生故障，另一个将无法达到法定人数[4]，往后的写入将会失败，直到第二个成员恢复为止。

表11-1：集群大小对容错的影响

服务器数量	达到大多数的数量	容错数量
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3

从表 11-1 中可以看到，增加成员的数目可以提升容错能力。然而，成员数量的增加也意味着在写入时需要在更多的节点之前进行商议和沟通，从而影响系统性能。由于必须有足够多的节点同时发生故障才会导致系统崩溃，一旦集群中的节点数目超过 7 个，出现这种情况的可能性就非常低，因此考虑到性能，就不值得再增加节点数目。还要注意成员数目一般最好避免偶数，因为集群节点的数量虽然增多了（并因此影响了性能），但并未提升容错能力。

当然，许多分布式系统的主机数目远远超过 7 个。这种情况下，其中 5 个或 7 个主机可以用于集群的组成，其余节点上运行的客户端可以只用作系统查询，不参与达成共识。这一点在 etcd 中以**代理**来实现，而在 Consul 中则以**客户端模式**实现。

首先利用 Docker machine 创建两个新主机：

```
$ docker-machine create -d virtualbox etcd-1
...
$ docker-machine create -d virtualbox etcd-2
...
```

现在可以启动 etcd 容器。因为已预先知道 etcd 集群的成员，所以只需在启动容器时明确列出它们便可以了。如果无法预知容器地址，也可以使用集群提供的基于 URL 或 DNS 的

注 4：简单来说就是“大多数”的意思。

发现机制。启动 etcd 的时候需要设置很多选项，因此我使用了环境变量来保存虚拟机的 IP 地址，这样做可以让启动简单些：

```
$ HOSTA=$(docker-machine ip etcd-1)
$ HOSTB=$(docker-machine ip etcd-2)
$ eval $(docker-machine env etcd-1)
$ docker run -d -p 2379:2379 -p 2380:2380 -p 4001:4001 \
  --name etcd quay.io/coreos/etcd:v2.2.5 \  ❶
  -name etcd-1 -initial-advertise-peer-urls http://${HOSTA}:2380 \  ❷
  -listen-peer-urls http://0.0.0.0:2380 \
  -listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
  -advertise-client-urls http://${HOSTA}:2379 \
  -initial-cluster-token etcd-cluster-1 \
  -initial-cluster \
     etcd-1=http://${HOSTA}:2380,etcd-2=http://${HOSTB}:2380 \  ❸
  -initial-cluster-state new
...
d4c12bbb16042b11252c5512ab595403fefcb2f46abb6441b0981103eb596eed
```

❶ 从 quay.io 寄存服务获取官方的 etcd 镜像。

❷ 设置访问 etcd 所需的各个 URL。为了能够建立来自远端和本地的连接，我们必须确保 etcd 监听 IP 地址 `0.0.0.0`，但要告诉其他客户端和 etcd 节点经由主机 IP 连接。在实际的设置中，etcd 节点应该只会在内部网络上通信，而不应把它暴露于外网（换句话说，不应通过 `docker-machine ip` 来获取 IP）。

❸ 明确列出集群中的所有节点，包括正在启动的节点。这部分信息可以用其他的服务发现方法替代。

第二个虚拟机的设置几乎一模一样，唯一的区别是需要对外部的客户端公布 `etcd-2` 的 IP 地址：

```
$ eval $(docker-machine env etcd-2)
$ docker run -d -p 2379:2379 -p 2380:2380 -p 4001:4001 \
  --name etcd quay.io/coreos/etcd:v2.2.5 \
  -name etcd-2 -initial-advertise-peer-urls http://${HOSTB}:2380 \
  -listen-peer-urls http://0.0.0.0:2380 \
  -listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
  -advertise-client-urls http://${HOSTB}:2379 \
  -initial-cluster-token etcd-cluster-1 \
  -initial-cluster \
     etcd-1=http://${HOSTA}:2380,etcd-2=http://${HOSTB}:2380 \
  -initial-cluster-state new
...
2aa2d8fee10aec4284b9b85a579d96ae92ba0f1e210fb36da2249f31e556a65e
```

现在我们的 etcd 集群已经启动了。我们可以利用 curl 给 etcd 的 HTTP API 发送一个简单的查询，要求列出所有成员的名单：

```
$ curl -s http://$HOSTA:2379/v2/members | jq '.'
{
  "members": [
    {
      "clientURLs": [
```

```
        "http://192.168.99.100:2379"
      ],
      "peerURLs": [
        "http://192.168.99.100:2380"
      ],
      "name": "etcd-1",
      "id": "30650851266557bc"
    },
    {
      "clientURLs": [
        "http://192.168.99.101:2379"
      ],
      "peerURLs": [
        "http://192.168.99.101:2380"
      ],
      "name": "etcd-2",
      "id": "9636be876f777946"
    }
  ]
}
```

为了使输出排列整齐，这里使用了一个名为 jq 的工具。通过发送 `POST` 和 `DELETE` 的 HTTP 请求到相同的端点，还可以在集群中动态添加或删除成员。

下一步添加一些数据，然后验证数据在两个不同的主机上都能够读取到。数据存储的位置是在 etcd 的目录下，并以 JSON 格式返回。下面的例子通过 HTTP `PUT` 请求将 `service_address` 的值存储在 service_name 目录下：

```
$ curl -s http://$HOSTA:2379/v2/keys/service_name \
        -XPUT -d value="service_address" | jq '.'
{
  "node": {
    "createdIndex": 17,
    "modifiedIndex": 17,
    "value": "service_address",
    "key": "/service_name"
  },
  "action": "set"
}
```

我们只需对该目录执行一个 `GET` 请求，就能把值取回来：

```
$ curl -s http://$HOSTB:2379/v2/keys/service_name | jq '.'
{
  "node": {
    "createdIndex": 17,
    "modifiedIndex": 17,
    "value": "service_address",
    "key": "/service_name"
  },
  "action": "get"
}
```

默认情况下，etcd 除了返回某个键的值，还会返回一些元数据。请注意，虽然我们在

etcd-1 上设置数据，但返回值是从 etcd-2 读取的。由于它们隶属同一个集群，对哪个主机操作都是可以的，它们给出的答案必定一致。

还有一个名为 etcdctl 的命令行客户端，它可用于与 etcd 集群交互。虽然我们可以直接安装它，但这里我选择使用容器：

```
$ docker run binocarlos/etcdctl -C ${HOSTB}:2379 get service_name
service_address
```

通过查看 etcd 容器的日志，你能看到很多 etcd 工作时的细节，其中包括成员之间如何协作选出 leader，这对你了解 etcd 的工作原理将很有帮助。关于 etcd 采用的 Raft 算法，可以在 https://raft.github.io/ 上看到完整的详细介绍。

现在我们应该已经知道怎样编写一个直接利用 etcd 作为服务发现的应用程序了。如果需要修改我们的 identidock 程序，只需要在 Python 代码中添加一个简单的 HTTP 请求即可，该请求用来查找 Redis 和 dnmonster 服务的地址。如果要更进一步的话，还可以修改 dnmonster 和 Redis 容器，使它们在启动时把自己的地址登记到 etcd 中，这样就实现全自动化了。

虽然可以通过修改 identidock 代码来增加服务发现功能，不过下一节将会介绍如何利用 SkyDNS 建立一个基于 etcd 的服务发现方案，而无需修改任何代码。

11.2.2 SkyDNS

SkyDNS（https://github.com/skynetservices/skydns）在 etcd 之上实现了一个基于 DNS 的服务发现方案。值得一提的是，Google Container Engine 正是采用了 SkyDNS 来实现 Kubernetes 的服务发现功能（参见 12.1.3 节）。

我们可以利用 SkyDNS 实现 etcd 解决方案，而且不需要改动任何代码，就能够把 identidock 分布在两台主机上运行。如果你已经跟着之前的例子一起做了，你现在应该已经有一个由两台服务器组成的 etcd 集群，一台是 IP 地址为 $HOSTA 的 etcd-1，另一台是 IP 地址为 $HOSTB 的 etcd-2。当完成这一节的例子后，我们的系统架构将如图 11-4 所示，其中的 identidock 容器将利用 SkyDNS 寻找 dnmonster 和 Redis 容器的所在位置。

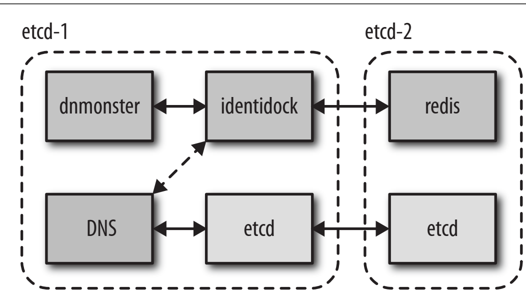


图 11-4：利用 SkyDNS 和 etcd 实现 identidock 的跨主机配置

首先要做的就是为 etcd 添加一些 SkyDNS 的配置，让 SkyDNS 启动时知道该做什么：

```
$ curl -XPUT http://${HOSTA}:2379/v2/keys/skydns/config \
      -d value='{"dns_addr":"0.0.0.0:53", "domain":"identidock.local."}' | jq .
{
  "action": "set",
  "node": {
    "key": "/skydns/config",
    "value": "{\"dns_addr\":\"0.0.0.0:53\", \"domain\":\"identidock.local.\"}",
    "modifiedIndex": 6,
    "createdIndex": 6
  }
}
```

上面的配置告诉 SkyDNS 监听所有网络接口的 53 端口，并作为 identidock.local 这个域名的权威服务器（authority）。

虽然 SkyDNS 可以直接在主机上运行，但使用 SkyDNS 的容器比较明智。我们将要使用的是由 SkyDNS 开发者创建的 skynetservices/skydns 镜像[5]。现在让我们在 etcd-1 上启动它：

```
$ eval $(docker-machine env etcd-1)
$ docker run -d -e ETCD_MACHINES="http://${HOSTA}:2379,http://${HOSTB}:2379" \
            --name dns skynetservices/skydns:2.5.2a
...
f95a871247163dfa69cf0a974be6703fe1dbf6d07daad3d2fa49e6678fa17bd9
```

在之前的例子中，我们需要提供一些额外的参数来告诉 etcd 后台的位置，但现在已经无需这样做了。我们已经有了一个可以工作的 DNS 服务器，只是还没有告诉它任何服务。现在让我们在 etcd-2 上启动 Redis 服务器，并把它添加到 SkyDNS：

```
$ eval $(docker-machine env etcd-2)
$ docker run -d -p 6379:6379 --name redis redis:3
...
d9c72d30c6cbf1e48d3a69bc6b0464d16232e45f32ec00dcebf5a7c6969b6aad
$ curl -XPUT http://${HOSTA}:2379/v2/keys/skydns/local/identidock/redis \
      -d value='{"host":"'$HOSTB'","port":6379}' | jq .
{
  "action": "set",
  "node": {
    "key": "/skydns/local/identidock/redis",
    "value": "{\"host\":\"192.168.99.101\",\"port\":6379}",
    "modifiedIndex": 7,
    "createdIndex": 7
  }
}
```

curl 请求的路径以 /local/identidock/redis 结尾，表示目标域名为 redis.identidock.local。参数中的 JSON 数据表示域名解析后的 IP 地址和端口。我们使用的 IP 地址是主机的 IP 地址而不是 Redis 容器的 IP，因为容器的 IP 地址是在 etcd-2 的本地网络。

现在可以先试试这个配置能否正常工作。我们需要启动一个新容器，并把 --dns 选项指向

注 5：撰写本书的时候，SkyDNS 镜像还没有利用自动构建生成，因此它就像一个黑盒子。你可能更希望自己构建一个 SkyDNS 容器，这样便可以确定里面究竟有什么东西。如果想这样做的话，在 SkyDNS 的 GitHub 项目（https://github.com/skynetservices/skydns）上有一个 Dockerfile 可供使用。

我们的 DNS 容器，让它负责域名查询：

```
$ eval $(docker-machine env etcd-1)
$ docker run --dns $(docker inspect -f {{.NetworkSettings.IPAddress}} dns) \
              -it redis:3 bash
...
root@3baff51314d6:/data# ping redis.identidock.local
PING redis.identidock.local (192.168.99.101): 48 data bytes
56 bytes from 192.168.99.101: icmp_seq=0 ttl=64 time=0.102 ms
56 bytes from 192.168.99.101: icmp_seq=1 ttl=64 time=0.090 ms
56 bytes from 192.168.99.101: icmp_seq=2 ttl=64 time=0.096 ms
^C--- redis.identidock.local ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.090/0.096/0.102/0.000 ms
root@3baff51314d6:/data# redis-cli -h redis.identidock.local ping
PONG
```

不错！唯一的问题是 redis.identidock.local 这个名字有点冗长。如果可以把它缩短为 redis 就好了，可是这样不行：

```
root@3baff51314d6:/data# ping redis
ping: unknown host
```

如果在启动新容器时加上 identidock.local 作为 DNS 搜索的域名，那么操作系统便会在 redis 无法解析的时候，自动尝试解析 redis.identidock.local：

```
root@3baff51314d6:/data# exit
$ docker run --dns $(docker inspect -f {{.NetworkSettings.IPAddress}} dns) \
             --dns-search identidock.local \
             -it redis:3 redis-cli -h redis ping
PONG
```

非常好，这和我们想要的相差不远了，但我们不希望每次运行容器时都要指定 `--dns` 和 `--dns-search` 选项。我们也可以在执行 Docker 守护进程时指定这两个选项，但是如果 DNS 服务器本身是一个容器的话，我们就遇到一个先有鸡还是先有鸡蛋的问题了 [6]，因此需要选择另一种方法。这个方法就是把域名等信息添加到主机的 /etc/resolv.conf 文件中 [7]，这个文件告诉操作系统如何寻找域名，而且其中的信息也会被传到容器中：

```
$ docker-machine ssh etcd-1
...
docker@etcd-1:~$ echo -e "domain identidock.local \nnameserver " \
  $(docker inspect -f {{.NetworkSettings.IPAddress}} dns) > /etc/resolv.conf
docker@etcd-1:~$ cat /etc/resolv.conf
domain identidock.local
nameserver 172.17.0.3
docker@etcd-1:~$ exit
```

注 6：如果你要这样做的话，可以将 DNS 容器的 53 端口开放给主机，并以 Docker 网桥的地址作为 DNS 服务器的地址。

注 7：VirtualBox 的虚拟机实际上会在重启的时候重新创建这个文件，等于把我们的修改还原，因此这里的操作仅作为示范之用。毕竟在生产环境中的主机，修改 resolv.conf 的方法不尽相同，例如通过 resolvconf 程序。

再试试看吧：

```
$ docker run redis:3 redis-cli -h redis ping
PONG
```

现在来启动 dnmonster，并把它添加到 DNS：

```
$ docker run -d --name dnmonster amouat/dnmonster:1.0
$ DNM_IP=$(docker inspect -f {{.NetworkSettings.IPAddress}} dnmonster)
$ curl -XPUT http://$HOSTA:2379/v2/keys/skydns/local/identidock/dnmonster \
      -d value='{"host": "'$DNM_IP'","port":8080}'
...
```

这里使用了 dnmonster 的内部容器 IP，因此它只能从 `etcd-1` 访问。如果我们有多个 SkyDNS 服务器在不同的主机上运行，为避免这个记录对其他服务器造成混淆，最好把它标记为 host local。你可以在启动 SkyDNS 时定义一个 host local 的域名。

最后，把所有 Docker 连接统统拿掉，然后启动 identidock，确认它能够正常工作：

```
$ docker run -d -p 80:9090 amouat/identidock:1.0
$ curl $HOSTA
<html><head><title>...
```

现在我们已经拥有一个服务发现接口了，它不仅不需要对原来的程序作任何修改，而且运行在一个具备容错能力的 etcd 分布式存储之上。

“挖掘”SkyDNS

如果想了解 SkyDNS 是如何工作的，可以利用 SkyDNS 镜像中包含的 dig 工具。例如：

```
$ docker exec -it dns sh
/ # dig @localhost SRV redis.identidock.local
dig @localhost SRV redis.identidock.local

; <<>> DiG 9.10.1-P2 <<>> @localhost SRV redis.identidock.local
; (2 servers found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 51805
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 1

;; QUESTION SECTION:
;redis.identidock.local.      IN  SRV

;; ANSWER SECTION:
redis.identidock.local. 3600  IN  SRV 10 100 6379 redis.identidock.local.

;; ADDITIONAL SECTION:
redis.identidock.local. 3600  IN  A 192.168.99.101

;; Query time: 4 msec
;; SERVER: ::1#53(::1)
;; WHEN: Sat Jul 25 17:18:39 UTC 2015
;; MSG SIZE rcvd: 98
```

DNS 服务器返回了 `redis.identidock.local` 的 SRV 记录。其中包括 IP 地址和端口号，以及优先级、权重和 TTL。

SkyDNS 使用了 SRV，或称之为服务（service）的记录，以及传统的 A 记录（用于解析 IPv4 地址）。SRV 是返回的众多记录中的一个，包含服务的端口号、存活时间（time-to-live，TTL）、优先级和权重。设置 TTL 的用途之一是为了确保当客户端或代理没有定时更新它时，记录能够被自动删除。它能够用于实现故障切换，以及比简单地让客户端超时更温和的错误处理办法。

SkyDNS 的其他功能还包括把多台主机集合成为一个地址池（address pool）用作负载均衡，以及把指标和统计数据发布至 Prometheus 和 Graphite 等服务。

11.2.3 Consul

Consul（https://consul.io）是 Hashicorp 公司对服务发现这个难题的回应。除了作为一个分布式且高度可用的键值存储，它还有先进的健康检查功能，并默认配备了 DNS 服务器。

CAP 定理

当研究键值存储和服务发现时，很容易碰到 CAP 定理（https://en.wikipedia.org/wiki/CAP_theorem），大意是一个分布式系统不可能同时满足一致性（Consistency）、可用性（Availability）和分区容忍（Partition tolerant）这三个特性。[8]

一个 AP 系统会把一致性放在首位，因此读写几乎不会失败（而且一般速度很快），但数据可能不会一直保持最新（在某些情况下可能返回旧数据）。一个 CP 系统会把一致性放在首位，因此写操作有时候可能失败，但是返回的数据一直是正确和最新的。

在实际操作中，它们之间的区别并不是那么明显，尤其是在使用 Consul 的时候。etcd 和 Consul 都是基于 CP 系统的 Raft 算法。然而，Consul 具有三种不同的模式 [default（默认）、consistent（一致）、stale（陈旧）]，能够在一致性和可用性之间提供不同程度的取舍。

任何运行 Consul 代理（agent）实例的主机，必须采用服务器模式或客户端模式。代理能够查询各种服务状态以及一般统计数据，如内存使用情况，从而降低客户端程序的复杂度。部分主机（通常是 3 个、5 个或 7 个，详见 11.2.1 节的辅助栏“最佳集群大小”）的代理运行在服务器模式下，负责写入和存储数据，并与其他服务器代理协作。客户端模式的代理则把请求转发至服务器代理。

Consul 的入门也很容易，尤其是使用 Docker 容器的话。以下的范例中将使用来自 Glider Labs（http://gliderlabs.com/）的容器。与之前一样，先来创建两个新的虚拟机：

注 8：有关这些术语的准确定义，参见“Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”（https://www.comp.nus.edu.sg/~gilbert/pubs/BrewersConjecture-SigAct.pdf）。

```
$ docker-machine create -d virtualbox consul-1
...
$ docker-machine create -d virtualbox consul-2
...
```

现在启动 Consul 容器。与之前一样，为了节省一些操作，我们将虚拟机的 IP 地址保存在环境变量中：

```
$ HOSTA=$(docker-machine ip consul-1)
$ HOSTB=$(docker-machine ip consul-2)
$ eval $(docker-machine env consul-1)
$ docker run -d --name consul -h consul-1 \
        -p 8300:8300 -p 8301:8301 -p 8301:8301/udp \
        -p 8302:8302/udp -p 8400:8400 -p 8500:8500 \
        -p 172.17.42.1:53:8600/udp \
        gliderlabs/consul agent -data-dir /data -server \  ❶
                   -client 0.0.0.0 \  ❷
                   -advertise $HOSTA -bootstrap-expect 2  ❸
...
ff226b3114541298d19a37b0751ca495d11fabdb652c3f19798f49db9cfea0dc
```

❶ 以服务器模式启动 Consul 代理，数据保存在 /data 目录下。

❷ 在指定的地址监听来自客户端的 API 请求。该地址默认为 127.0.0.1，并只适用于容器内。

❸ `-advertise` 选项指定其他主机联系这个服务器时应使用的地址，这里指定的是主机的 IP 地址。此外还设置了 `-bootstrap-expect` 选项来告诉 Consul 需要等待第二个服务器加入集群。

刚才的命令是通过由 Docker machine 返回的公网 IP 连接各个主机的。在生产环境中，你使用的应该是一个无法从互联网访问得到的私有地址。

现在启动第二个容器，这次需要使用 `-join` 命令与第一台服务器连接：

```
$ eval $(docker-machine env consul-2)
$ docker run -d --name consul -h consul-2 \
        -p 8300:8300 -p 8301:8301 -p 8301:8301/udp \
        -p 8302:8302/udp -p 8400:8400 -p 8500:8500 \
        -p 172.17.42.1:53:8600/udp \
        gliderlabs/consul agent -data-dir /data -server \
              -client 0.0.0.0 \
              -advertise $HOSTB -join $HOSTA
...
```

通过 Consul 的命令行工具，能够确认容器已被添加到集群中：

```
$ docker exec consul consul members
Node      Address             Status  Type    Build  Protocol  DC
consul-1  192.168.99.100:8301  alive   server  0.5.2  2         dc1
consul-2  192.168.99.101:8301  alive   server  0.5.2  2         dc1
```

可以尝试对 Consul 设置和读取一些数据，了解它作为一个键值存储是如何工作的：

```
$ curl -XPUT http://$HOSTA:8500/v1/kv/foo -d bar
true
$ curl http://$HOSTA:8500/v1/kv/foo | jq .
[
```

```
  {
    "Value": "YmFy",
    "Flags": 0,
    "Key": "foo",
    "LockIndex": 0,
    "ModifyIndex": 39,
    "CreateIndex": 16
  }
]
```

看起来有点奇怪，数据的确是取回来了，但其中的值为什么是 "Value":"YmFy" 呢？原来，Consul 会动态地对数据进行 base64 编码。我们可以通过 jq 和 base64 命令把原始数据取回来：[9]

```
$ curl -s http://$HOSTA:8500/v1/kv/foo | jq -r '.[].Value' | base64 -d
bar
```

虽然操作稍微复杂一点，但总算是成功了。

Consul 还有另一套单独的 API 用于添加服务，它是 Consul 的服务发现和健康检查功能的一部分。一般情况下，键值存储只用于存储配置信息和少量元数据。

接下来看看如何通过 Consul 服务使 identidock 实现跨主机工作。我们的目标是保持配置与之前一样，即 Redis 运行在 consul-2 上，而 identidock 和 dnmonster 则运行在 consul-1 上。首先启动 Redis：

```
$ eval $(docker-machine env consul-2)
$ docker run -d -p 6379:6379 --name redis redis:3
...
2f79ea13628c446003ebe2ec4f20c550574c626b752b6ffa3b70770ad3e1ee6c
```

现在通过 /service/register 端点告诉 Consul 我们的 Redis 服务：

```
$ curl -XPUT http://$HOSTA:8500/v1/agent/service/register \
       -d '{"name": "redis", "address":"'$HOSTB'","port": 6379}'
$ docker run amouat/network-utils dig @172.17.42.1 +short redis.service.consul
...
192.168.99.101
```

接下来，为了使 Consul 能够用作 DNS 解析，需要对 consul-1 进行配置。这一次采取与之前的 etcd 例子不一样的方法，我们不会改动主机上的 /etc/resolv.conf 文件，而是配置 Docker 守护进程。为此，需要修改 /var/lib/boot2docker/profile 文件，并把 --dns 和 --dns-search 选项加进去：

```
$ docker-machine ssh consul-1
...
docker@consul-1:~$ sudo vi /var/lib/boot2docker/profile
...
docker@consul-1:~$ cat /var/lib/boot2docker/profile

EXTRA_ARGS='
--label provider=virtualbox
--dns 172.17.42.1
--dns-search service.consul  ❶
```

注 9：这里使用的 base64 是 GNU Linux 的版本。如果你使用的是 MacOS 的版本，那么参数 -d 需要改为 -D。

```
'
CACERT=/var/lib/boot2docker/ca.pem
DOCKER_HOST='-H tcp://0.0.0.0:2376'
DOCKER_STORAGE=aufs
DOCKER_TLS=auto
SERVERKEY=/var/lib/boot2docker/server-key.pem
SERVERCERT=/var/lib/boot2docker/server.pem
```

❶ 这个选项允许缩短域名，例如以“redis”代替全名“redis.service.consul”。

现在需要重新启动守护进程并启动 Consul。我认为最简单的方法就是重启虚拟机：

```
docker@consul-1:~$ exit
$ eval $(docker-machine env consul-1)
$ docker start consul
consul
```

快速测试一下：

```
$ docker run redis:3 redis-cli -h redis ping
PONG
```

在 `consul-1` 上启动 dnmonster，并添加服务：

```
$ docker run -d --name dnmonster amouat/dnmonster:1.0
...
41c8a78989803737f65460d75f8bed1a3683ee5a25c958382a1ca87f27034338
$ DNM_IP=$(docker inspect -f {{.NetworkSettings.IPAddress}} dnmonster)
$ curl -XPUT http://$HOSTA:8500/v1/agent/service/register \
       -d '{"name": "dnmonster", "address":"'$DNM_IP'","port": 8080}'
```

最后，把 identidock 运行起来：

```
$ docker run -d -p 80:9090 amouat/identidock:1.0
...
22cfd97bfba83dc31732886a4f0aec51e637b8c7834c9763e943e80225f990ec
$ curl $HOSTA
<html><head><title>...
```

与之前一样，identidock 已经无需使用 Docker 连接了。

Consul 最有趣的功能之一称为*健康检查*，这个功能确保系统的各个部分都是活跃的，并且正常运作。我们可以针对主机节点本身或特定的服务编写测试程序（例如检查磁盘空间或内存）。下面的代码定义了一个针对 dnmonster 服务的简单 HTTP 测试：

```
$ curl -XPUT http://$HOSTA:8500/v1/agent/service/register \
       -d '{"name": "dnmonster", "address":"'$DNM_IP'","port": 8080,
            "check": {"http": "http://'$DNM_IP':8080/monster/foo",
                      "interval": "10s"}
          }'
```

这个测试的目的是为了确保当容器收到某个 URL 的 HTTP 请求时，能够返回 `2xx` 的状态码。注意，这个测试必须运行在 `consul-1` 上才能成功。可以通过访问 `/health/checks/dnmonster/` 端点来检查测试的状态：

```
$ curl -s $HOSTA:8500/v1/health/checks/dnmonster | jq '.[].Status'
"passing"
```

测试还可以用脚本实现，脚本的返回值为 0 则代表测试通过。使用脚本的好处是能够实现复杂的测试。把健康检查结合 Consul 的监视（watch）功能（用于检查数据更新）一起使用，就能够很容易地实现故障切换，并在问题发生时自动通报管理员。

Consul 还有其他主要功能，其中包括多数据中心的支持和网络流量加密。

11.2.4　服务注册

在前面的范例中，服务发现的最后一步“服务注册”是手动进行的，必须调用 curl 发送请求，把 Redis 和 dnmonster 注册到 SkyDNS 和 Consul。其实可以把这个逻辑放入 Redis 和 dnmonster 容器中，使它们在启动时自动进行注册[10]，还可以创建一个负责监听 Docker 事件的服务，当它发现容器启动时，便会自动进行容器注册。

这就是 Glider Labs 开发 Registrator（https://github.com/gliderlabs/registrator）的目的。Registrator 能够与 Consul、etcd 或 SkyDNS 配合使用，实现容器的自动注册。它的工作原理是监听 Docker 事件流中的容器创建事件，并按容器的元数据添加相关条目到该容器所使用的框架。

基于 DNS 的服务发现的优缺点

这里介绍的许多解决方案都为实现服务发现功能提供了 DNS 接口。DNS 在一部分解决方案中是主要的，甚至是唯一的接口；但在另一些方案中，它可能只是服务本身所有 API 的一部分，仅为了方便而提供。

服务发现偏向选择 DNS 有以下几个主要原因。

- DNS 能够直接支持古老的应用程序。在使用其他非 DNS 的机制时，虽然可以利用大使容器来解决这个问题，但需要开发和运营团队耗费额外的精力。
- 开发者无需做任何特别的事情，也不用学习新的 API。使用 DNS 的应用程序，无需修改就能够在很多不同的平台上运行。
- DNS 是一个为人熟悉和可靠的协议，已有多种实现并得到了广泛支持。

基于 DNS 的服务发现也有一些缺点，因此在某些情况下你可能会考虑其他方案。一般 DNS 被批评速度慢，对于远程查询的确如此，但对我们来说却问题不大，因为服务发现的使用场景都是本地查询，速度其实很快。下面是我们要注意的其他问题。

- 普通的DNS 查询不会返回端口信息，因此要么自己假定，要么通过其他渠道获得（DNS SRV 记录确实会返回端口信息，但几乎所有应用程序和框架都只会使用主机名）。
- 应用程序（包括操作系统）可能会缓存 DNS 的响应结果，因此有可能出现服务被迁移后，客户端还没有更新它的 DNS 记录的情况。
- 大多数的 DNS 服务只提供健康检查和负载均衡的有限支持。通常负载均衡的做法仅限于循环（round-robin）或随机选择，因此只能满足部分用户。健康检查可以利用 TTL 实现，但更复杂和更深入的检查通常都需要更多的服务来完成。
- 客户端可以自行决定使用哪个服务，选择标准基于 API 版本或容量等属性。

注 10：如果我们不希望或无法修改容器内的服务，可以通过一个封装脚本或“side-kick”进程来完成。

11.2.5 其他解决方案

下面还有许多其他服务发现方案可供选择。

ZooKeeper（https://zookeeper.apache.org/）

ZooKeeper 是一个可靠的集中式存储，它作为一个现成的软件，已被 Mesos 和 Hadoop 用于服务协调。ZooKeeper 用 Java 编写，使用时需要利用 Java API。如果你不用 Java，现在也有数种语言的绑定（binding）可用。客户端需要与 ZooKeeper 服务器维持活跃连接，并保持 keep-alive 的动作，因此需要不少的编程工作 [但有一些程序库，例如 Curator（https://curator.apache.org），可以帮助你来完成这个工作]。

ZooKepper 的主要优点是成熟、稳定且千锤百炼。如果你已经有基础设施正在使用 ZooKeeper，它不失为一种不错的选择。否则，你可能不值得把精力花在 ZooKeeper 的集成和开发工作上，尤其是如果你并没有使用 Java。

SmartStack（http://nerds.airbnb.com/smartstack-service-discovery-cloud/）

SmartStack 是 Airbnb 的服务发现解决方案。它由两部分组成：负责健康检查和注册的 Nerve（https://github.com/airbnb/nerve），以及负责处理服务发现的 Synapse（https://github.com/airbnb/synapse）。

Synapse 运行于每个需要使用服务的主机上，并且把每个服务分配到一个端口，代理会把端口对应到实际的服务。Synapse 利用 HAProxy（http://www.haproxy.org/）负责路由，当有任何变化发生时，它便会自动更新 HAProxy 并把它重新启动。你可以设置 Synapse，让它获取需要运行代理的服务列表，可以从存储中（例如 ZooKeeper 或 etcd）或通过监视 Docker 的事件流中出现的容器创建事件（类似 Registrator）来获取。

每个服务都会有一个相应的 Nerve 进程或容器，负责检查服务的健康，以及自动注册到 Synapse 所使用的存储（例如 ZooKeeper 或 etcd）。

Eureka（https://github.com/Netflix/eureka/wiki）

Eureka 是 Netflix 用于 AWS 的负载均衡和故障切换方案。它的设计是一个“中间层”方案，以应对 AWS 节点生命周期短暂的特性。如果你打算在 AWS 的基础设施上运行大规模的服务，它绝对是值得研究的方案。

WeaveDNS（https://www.weave.works/docs/net/latest/weavedns/）

WeaveDNS 是 Weave 联网方案中的服务发现组件。容器启动的时候会把它们自己的主机或容器名称注册到 WeaveDNS，这样就能提供一套完整的自动化方案。WeaveDNS 作为 Weave 路由器的一部分运行在每台主机上，通过与网络上的其他 Weave 路由器互相通信，因此它能够解析所有容器的名称。WeaveDNS 还提供了简单的负载均衡。更多信息参见 11.5.2 节。

docker-discover（http://jasonwilder.com/blog/2014/07/15/docker-service-discovery/）

基本上它是 SmartStack 以 Docker 实现的版本，以 etcd 作为后端。与 SmartStack 一样，它由两部分组成：一个是 docker-register（https://github.com/jwilder/docker-register），

相当于 Nerve 的健康检查和注册组件；另一个是 docker-discover，相当于 Synapse 负责服务发现的组件（https://github.com/jwilder/docker-discover）。docker-discover 与 SmartStack 一样，都是使用 HAProxy 处理路由。虽然它是一个非常有趣的项目，但由于缺乏更新和组织支持，可能只会有零星的开发活动和支持。

最后值得一提的是，Docker 即将推出全新的联网功能（参见 11.4 节），通过服务对象（service object）提供有限的服务发现。而这种服务发现依赖于 Docker 的 Overlay 联网驱动程序，或其他兼容插件，如 Calico。

11.3 联网选项

正如前文所述，只要底层的网络允许，无论是通过大使容器还是服务发现方案，都可以让服务实现跨主机连线。然而，这需要在主机上公开端口，涉及手动管理，因此不利于扩展。一个更好的解决方法是，在容器之间提供 IP 连接，而这正是本章将要介绍的方案中的重点。

不过，在了解如何实现一套完整的跨主机联网方案之前，有必要先了解默认的 Docker 联网的工作原理，以及其中有哪些选项可用。Docker 有四个可用的基本模式：*网桥*（bridge）、*主机*（host）、*容器*（container）和*未联网*（none）。

11.3.1 网桥模式

默认的网桥网络在开发阶段中非常有用，它提供了一个无痛的方式，让容器能够互相交谈。但它并不太适用于生产环境，因为在网桥背后涉及很多底层工作，会导致相当大的开销。

图 11-5 显示的是使用 Docker 网桥时的网络架构。图中有一个用于连接各个容器的 Docker 网桥，它的名字通常是 `docker0`，地址一般为 `172.17.42.1`。容器启动后，Docker 会生成一对 veth 接口，本质上相当于软件实现的以太网物理连接，Docker 通过 veth 接口把容器的 `eth0` 连接到网桥。外部连接可以通过 IP 伪装（IP masquerading）的方式提供，IP 伪装是网络地址转换（NAT）的一种方式，以 IP 转发（IP forwarding）和 iptables 规则建立。

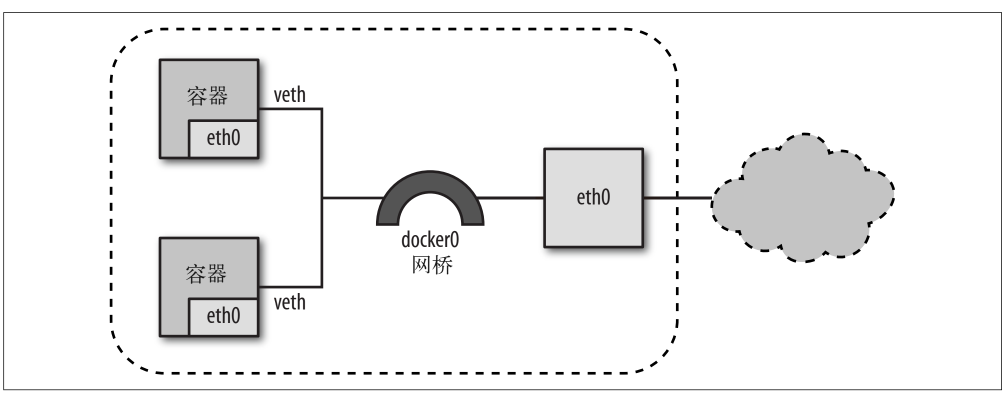


图 11-5：默认的 Docker 网桥联网

默认情况下，所有容器都可以互相沟通，无论是否已被连接，也无论是否已导出或公开端口（参见 13.7.2 节中的例子）。为了禁止这个行为，可以在 Docker 守护进程启动的时候加上 `--icc=false` 参数，这个参数将设置一个 iptables 规则，把容器之间的通信关闭。如果同时设置了 `--icc=false` 和 `--iptables=true`，那么只有已连接的容器才能通信，这也是通过 iptables 规则实现的。

以上所讲的都很适合开发阶段，但如果考虑效率的话，或许对生产环境就不太适合了。

11.3.2 主机模式

如果容器以 `--net=host` 参数启动，那么它便会共享主机的网络命名空间，还会把自己暴露在公网之上。这意味着容器与主机必须共用同一个 IP 地址，不过这就减少了网桥模式中涉及的底层开销，因此速度与常规的主机网络一样快。

由于 IP 地址是共享的，需要互相通信的容器必须预先协定使用哪些端口通信，在进行配置的时候必须考虑这一点，而且可能还需要修改程序。

这个模式也存在安全隐患，你可能会无意中把某些端口暴露于外界，不过可以使用防火墙进行监控。

由于这个模式的效率有显著的提升，你可以考虑使用混合的联网模型，其中面向外部网络和网络流量大的容器（譬如代理和缓存）可以使用主机模式，而其他容器则使用内部网络的网桥模式。请注意，网桥网络上的容器无法使用 Docker 连接与主机网络上的容器互联，但可以通过 `docker0` 网桥的 IP 地址互通。

11.3.3 容器模式

这个模式使用另一个容器的网络空间，在某些情况下可以很有用（例如，当你已有一个配置好网络栈的容器，而你想告诉其他容器使用这个栈时）。这个模式便于创建和重复使用专为某特定场景或数据中心架构而建立的高效网络栈。但它的缺点是，所有共享同一网络栈的容器将使用相同的 IP 地址。

它在某些情况下表现出色，而且为 Kubernetes 所采用（参见 12.1.3 节）。

11.3.4 未联网模式

顾名思义，这个模式意味着把容器的网络完全关闭。它适合无需任何联网的容器，例如一个只是把编译结果写到数据卷的编译器容器。

如果你打算从零开始搭建你的网络架构，那么未联网模式就非常有用了。如果你正准备这样做，可以参考像 pipework（https://github.com/jpetazzo/pipework）这样的工具，它对处理 cgroup 和命名空间的网络设置非常有用。

11.4 全新的Docker联网功能

撰写本书的时候，Docker 联网栈和接口正在进行大规模检修。到本书付印时，修改很可

能已经完成，它将会为 Docker 网络的创建和使用带来重大改变（虽然书中的代码应该能继续工作）。这一节会根据 Docker 的试验版来了解其中将会发生的变化。由于修改尚未完成，这里介绍的内容可能与正式版本发布时有些许出入。

对我们而言，影响最直接的改变就是将会有两个全新的最上层“对象”，分别是网络（network）和服务（service）。这使得“网络”[11] 的创建和管理从容器中抽离。当容器启动时，它可以被分配到某个网络，但只能与同一网络中的其他容器直接联系。容器可以发布服务，并允许通过名称寻找它，从而取代使用 Docker 连接的必要（虽然 Docker 连接仍然可用，但用途将会减少）。

举一些例子就更好理解了。

`network ls` 子命令可以列出当前网络及其 ID：

```
$ docker network ls
NETWORK ID          NAME                TYPE
d57af6043446        none                null
8fcc0afef384        host                host
30fa18d442b5        bridge              bridge
```

启动容器的时候，可以通过 `--publish-service` 参数创建服务。以下的命令在网桥网络中会创建一个 `db` 服务：

```
$ docker run -d --name redis1 --publish-service db.bridge redis
9567dd9eb4fbd9f588a846819ec1ea9b71dc7b6cbd73ac7e90dc0d75a00b6f65
```

通过 `service ls` 子命令可以查看现有的服务：

```
$ docker service ls
SERVICE ID          NAME                NETWORK             CONTAINER
f87430d2f240        db                  bridge              9567dd9eb4fb
```

现在我们在同一网络创建一个新容器，通过“db”服务便可以连接至 `redis1` 容器，而无需再使用 Docker 连接：

```
$ docker run -it redis redis-cli -h db ping
PONG
```

属于不同网络的容器默认无法通信。

如果使用 Docker 连接，可以看到它其实是在背后通过建立服务实现的：

```
$ docker run -d --name redis2 redis
7fd526b2c7a6ad8a3faf4be9c1c23375dc5ae4cd17ff863a293c67df816a2b09
$ docker run --link redis2:redis2 redis redis-cli -h redis2 ping
PONG
$ docker service ls
SERVICE ID          NAME                NETWORK             CONTAINER
59b749c7fe0b        redis2              bridge              7fd526b2c7a6
f87430d2f240        db                  bridge              9567dd9eb4fb
```

注 11：个人认为“网络”一词容易产生误导。在许多方面，Docker 的网络实际上只是容器的命名空间，可以实施分组和隔离，以及提供通信渠道。

更有趣的是，我们可以通过 `service attach` 和 `service detach` 两个子命令来重新分配提供服务的容器：

```
$ docker run redis redis-cli -h db set foo bar  ❶
OK
$ docker run redis redis-cli -h redis2 set foo baz  ❷
OK
$ docker run redis redis-cli -h db get foo
bar
$ docker service detach redis1 db
$ docker service attach redis2 db
$ docker run redis redis-cli -h db get foo  ❸
baz
```

❶ 往 `redis1` 添加数据，它目前正提供 `db` 服务。
❷ 往 `redis2` 添加数据。
❸ 现在 `db` 服务已由 `redis2` 提供。

于是可以看到，新模型在容器连接和系统维护方面提供了更高的灵活性和更多的监控手段，同时还支持以前的网络连接方式。

网络类型和插件

你可能注意到了网络具有不同*类型*。[12]“传统”的网络模式，包括之前介绍过的主机、未联网及网桥模式，它们各自都有一种类型。除此之外，还有 overlay 类型。通过*联网插件*，还可以添加新的类型。默认使用的网络可以在 Docker 守护程序中设置。如果没有设置默认网络，那就使用网桥网络。

插件

添加 Docker 插件，包括网络驱动在内[13]，只需安装到 /usr/share/docker/plugins 目录下即可。一般的做法是运行一个挂载这个目录的容器。

插件可以用任何语言编写，只要它们能够与 Docker 的 JSON-RPC（http://www.jsonrpc.org/）API 互通。

我们将在 11.5.4 节中看到一个使用 Project Calico 网络插件的例子。我期待未来将会出现各式各样使用不同底层技术 [如 IPVLAN（https://github.com/torvalds/linux/blob/master/Documentation/networking/ipvlan.txt） 和 Open vSwitch（http://openvswitch.org/）] 的插件，以应付不同的场景。

11.5 网络解决方案

下面将会介绍用于实施跨主机联网的各种容器集群方案。其中包括 Overlay，它是 Docker

注 12：也被称为*网络驱动*。

注 13：新的*数据卷*驱动也可以通过插件加载，例如 flocker（https://github.com/ClusterHQ/flocker-docker-plugin）。写到这里的时候，最上层的数据卷对象的创建工作正在进行，但仍需要一段时间才能取得成果。

在未来的联网栈中自带的所谓“内附电池”方案；还有功能丰富且便于使用的 Weave、来自 CoreOS 的 Flannel，以及 Metaswitch 公司基于第 3 层网络的 Calico 项目。

Docker 的联网功能还处于非常初级的阶段，而且仍在不断变化。在它之上有一个非常广阔、潜力巨大的发展空间，有待各式各样的解决方案出现以针对不同的使用场景。虽然本书写作之际，这里介绍的工具都是最前沿的，但是这个领域变化极快，因此到本书出版时，这些方案的使用方法会有所改变，而新的解决方案也将应运而生。在选择使用哪个方案之前，你应该先对目前可用的方案进行研究，之后再作决定。

11.5.1 Overlay

Overlay 是 Docker 为实现跨主机联网而开发的“内附电池”方案。它使用 VXLAN 隧道来连接主机，主机有自己的 IP 地址空间，外网的连接通过 NAT 实现。同级之间的通信采用了 serf（https://serfdom.io/）程序库。

连接容器至 Overlay 网络的方法与标准的网桥网络大致相同，它们同样需要建立起 Linux 网桥，并需要一对 veth 接口用于容器之间互相连接。

注意，这里使用的是试验版

范例中的虚拟机使用的 Docker 为试验版，因此它与你使用的版本必定有些差异。当你阅读本书的时候，Docker 的稳定版已经能够支持网络插件，你应该使用它。

范例中使用的 Docker 和 Consul 版本如下：

```
docker@overlay-1:~$ docker --version
Docker version 1.8.0-dev, build 5fdc102, experimental
docker@overlay-1:~$ docker run gliderlabs/consul version
Consul v0.5.2
Consul Protocol: 2 (Understands back to: 1)
```

在下面的范例中，我用试验版分支里的 Docker 部署了两台主机，分别是 `overlay-1` 和 `overlay-2`，并以 Consul 作为键值存储。我们将采用与之前相同的架构部署 identidock，即 Redis 运行于 `overlay-2`，而 dnmonster 和 identidock 容器则运行于 `overlay-1`。

本书出版之时，稳定版应该也能够基本做到这些。为了确保 Docker 客户端和守护进程的版本一致，我已用 `ssh` 进入虚拟机检查过。

以用 `ssh` 进入 `overlay-2` 作为开始：

```
$ docker-machine ssh overlay-2
...
```

首先要做的便是以 `overlay` 驱动创建一个名为“ovn”的新网络：

```
docker@overlay-2:~$ docker network create -d overlay ovn
5d2709e8fd689cb4dee6acf7a1346fb563924909b4568831892dcc67e9359de6
```

```
docker@overlay-2:~$ docker network ls
NETWORK ID          NAME                TYPE
f7ae80f9aa44        none                null
1d4c071e42b1        host                host
27c18499f9e5        bridge              bridge
5d2709e8fd68        ovn                 overlay
```

通过 `network info` 子命令可以获得更多有关这个网络的详细信息：

```
docker@overlay-2:~$ docker network info ovn
Network Id: 5d2709e8fd689cb4dee6acf7a1346fb563924909b4568831892dcc67e9359de6
Name: ovn
Type: overlay
```

现在是时候启动 Redis 了，我们在命令中加上 `--publish-service redis.ovn` 参数，代表容器在“ovn”网络上发布一个名为“redis”的服务：

```
docker@overlay-2:~$ docker run -d --name redis-ov2 \
                               --publish-service redis.ovn redis:3
...
29a02f672a359c5a9174713418df50c72e348b2814e88d537bd2ab877150a4a5
```

如果现在退出 `overlay-2` 并通过 `ssh` 进入 `overlay-1`，可以看到它能访问相同的网络：

```
docker@overlay-2:~$ exit
$ docker-machine ssh overlay-1
docker@overlay-1:~$ docker network ls
NETWORK ID          NAME                TYPE
7f9a4f144131        none                null
528f9267a171        host                host
dfec33441302        bridge              bridge
5d2709e8fd68        ovn                 overlay
```

以同样的方式启动 dnmonster 和 identidock 容器，使用 `--publish-service` 连接到 `ovn` 网络：

```
docker@overlay-1:~$ docker run -d --name dnmonster-ov1 \
                        --publish-service dnmonster.ovn amouat/dnmonster:1.0
...
37e7406613f3cbef0ca83320cf3d99aa4078a9b24b092f1270352ff0e1bf8f92
docker@overlay-1:~$ docker run -d --name identidock-ov1 \
                        --publish-service identidock.ovn amouat/identidock:1.0
...
41f328a59ff3644718b8ce4f171b3a246c188cf80a6d0aa96b397500be33da5e
```

最后检查是否一切正常工作：

```
docker@overlay-1:~$ docker exec identidock-ov1 curl -s localhost:9090
<html><head><title>Hello...
```

我们只需执行简单的步骤，便能快速地在两台主机上实现 identidock 的跨主机配置。由于 Overlay 驱动的实现细节和使用方法在未来可能会有所改变，因此有关它的介绍暂且告一段落。

11.5.2 Weave

Weave（https://weave.works/）是一个开发者友好的联网解决方案，它的设计理念是希望开发者用最少的精力就能把它应用在各种不同的环境中。Weave 也许是现今最完整的方案，因为它包含了用于服务发现和负载均衡的 WeaveDNS，自带 IP 地址管理系统（IP address management，IPAM），并支持加密通信。

接下来看看利用 Weave 把 identidock 运行在两台主机上有多么容易。我们将沿用大使容器和服务发现的范例中曾经使用过的架构，即 Redis 运行在其中一台主机（`weave-redis`）上，而 identidock 和 dnmonster 容器则运行在另一台主机（`weave-identidock`）上。同样，我们将利用 Docker machine 为我们提供虚拟机。范例中使用的 Weave 版本是 `1.1.0`，如果你使用的版本更新，那么操作上的些许差异将在所难免。

首先来构建 `weave-redis` 虚拟机：

```
$ docker-machine create -d virtualbox weave-redis
...
```

然后 `ssh` 进去并安装 Weave：

```
$ docker-machine ssh weave-redis
...
docker@weave-redis:~$ sudo curl -sL git.io/weave -o /usr/local/bin/weave
docker@weave-redis:~$ sudo chmod a+x /usr/local/bin/weave
docker@weave-redis:~$ weave launch
Setting docker0 MAC (mitigate https://github.com/docker/docker/issues/14908)
Unable to find image 'weaveworks/weaveexec:v1.1.0' locally
v1.1.0: Pulling from weaveworks/weaveexec
...
Digest: sha256:8b5e1b692b7c2cb9bff6f9ce87360eee88540fe32d0154b27584bc45acbbef0a
Status: Downloaded newer image for weaveworks/weaveexec:v1.1.0
Unable to find image 'weaveworks/weave:v1.1.0' locally
v1.1.0: Pulling from weaveworks/weave
Digest: sha256:c34b8ee7b72631e4b7ddca3e1157b67dd866cae40418c279f427589dc944fac0
Status: Downloaded newer image for weaveworks/weave:v1.1.0
```

上面的命令首先从网站下载 Weave 程序，然后下载数个提供 Weave 基础设施的容器并启动它们。稍后将深入讨论每个容器负责的细节。

下一步会把 Docker 客户端从原来使用 Docker 守护进程改为 Weave 代理。这样做是为了在容器启动时能够让 Weave 建立一些用于联网的 hook：

```
docker@weave-redis:~$ eval $(weave env)
```

现在可以启动 Redis 容器了，它将自动连接到 Weave 的网络：

```
docker@weave-redis:~$ docker run --name redis -d redis:3
Unable to find image 'redis:3' locally
3: Pulling from redis
...
3c97d635be5107f5a79cafe3cfaf1960fa3d14eec3ed5fa80e2045249601583f
docker@weave-redis:~$ exit
```

接下来就要启动 identidock 和 dnmonster 的主机了。这一次选择通过 docker-machine 执行 ssh 命令，而不是自行登录进去，这样配置会容易一些。首先创建 weave-identidock 虚拟机并安装 Weave：

```
$ docker-machine create -d virtualbox weave-identidock
...
$ docker-machine ssh weave-identidock \
   "sudo curl -sL https://git.io/weave -o /usr/local/bin/weave && \
    sudo chmod a+x /usr/local/bin/weave"
```

这一次，当执行 weave launch 的时候，我们需要提供 weave-redis 主机的 IP：

```
$ docker-machine ssh weave-identidock \
      "weave launch $(docker-machine ip weave-redis)"
Unable to find image 'weaveworks/weaveexec:v1.1.0' locally
v1.1.0: Pulling from weaveworks/weaveexec
...
Digest: sha256:8b5e1b692b7c2cb9bff6f9ce87360eee88540fe32d0154b27584bc45acbbef0a
Status: Downloaded newer image for weaveworks/weaveexec:v1.1.0
Unable to find image 'weaveworks/weave:v1.1.0' locally
v1.1.0: Pulling from weaveworks/weave
Digest: sha256:c34b8ee7b72631e4b7ddca3e1157b67dd866cae40418c279f427589dc944fac0
Status: Downloaded newer image for weaveworks/weave:v1.1.0
```

现在应该检查一下联网是否成功。我们会 ssh 进入 weave-identidock，并测试能否访问在 weave-redis 上运行的 Redis 容器。请注意，为了使用 Weave 代理，我们同样需要设置环境变量：

```
$ docker-machine ssh weave-identidock
...
docker@weave-identidock:~$ eval $(weave env)
docker@weave-identidock:~$ docker run redis:3 redis-cli -h redis ping
...
PONG
```

成功了！在完成这个范例之前，启动 dnmonster 和 identidock 容器，确保应用能够如常运行：

```
docker@weave-identidock:~$ docker run --name dnmonster -d amouat/dnmonster:1.0
...
1bc9cdd5c3dd532d4f6a56529be8e2a068a9402c1e07df69ec33971f5c4b89b9
docker@weave-identidock:~$ docker run --name identidock -d -p 80:9090 \
    amouat/identidock:1.0
...
9b5e9c89a7807bcad2cff49dc0692d0e8d064494288df5405a6573d886c0208d
docker@weave-identidock:~$ exit
$ curl $(docker-machine ip weave-identidock)
<html><head>...
$ curl -s $(docker-machine ip weave-identidock)/monster/gordon | head -c 4
�PNG
```

太好了，我们成功实现了跨主机的联网部署，而且容器名称都是通过 DNS 来解析的，而最重要的是没有使用任何 Docker 连接。

为了理解 Weave 如何运作，你必须对 Weave 启动的后台管理容器有所了解：

```
$ docker ps
CONTAINER ID ... PORTS                                              NAMES
0b7693194bb9                                                        weaveproxy
b6e515f4d02b     172.17.42.1:53->53/udp, 0.0.0.0:6783->6783/t...   weave
```

图 11-6 展示了 identidock 及以上的 Weave 管理容器。

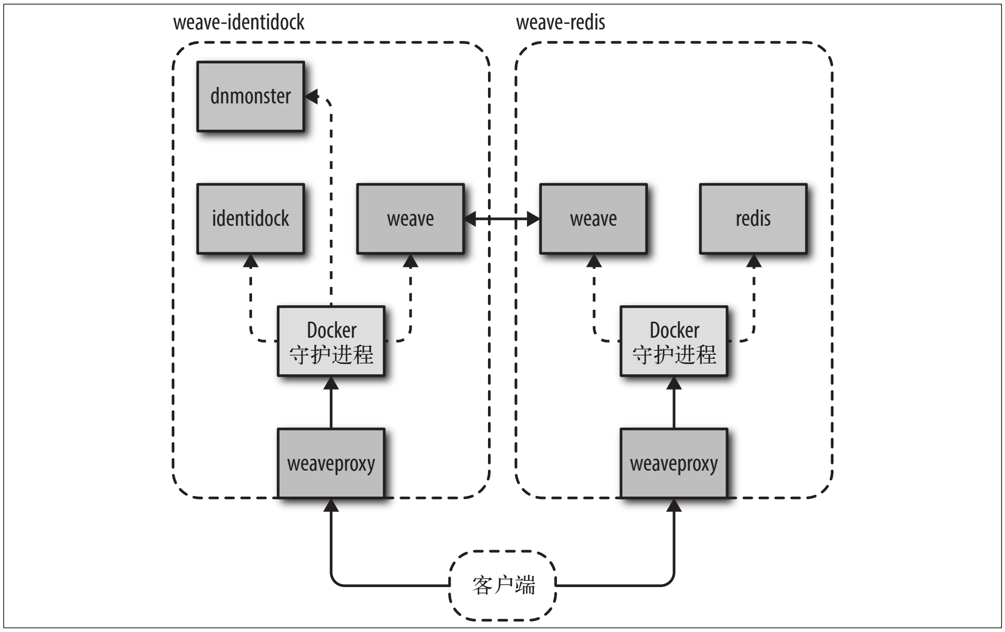


图 11-6：运行在 Weave 之上的 identidock

这两台主机运行着相同的 Weave 镜像，它们都是由先前的 `weave launch` 命令启动的。这些容器负责 Weave 基础设施的不同部分。

weave

这个容器包含 Weave 路由器，它负责处理网络路由，以及在 Weave 网络上与其他主机通信。Weave 路由器通过 TCP 与其他主机建立通信以及传递网络拓扑信息。网络流量则通过另外建立的 UDP 连接传送。Weave 路由器随着时间能够学习网络拓扑，使它们能有效地传送数据，以及在无需与所有主机建立连线的情况下也能够应对不断变化的网络架构。路由器还可以处理 DNS 请求，使开发者可以用名字来称呼其他主机上的容器。

weaveproxy

这个容器的神奇之处在于，用户只需如常执行 `docker run` 命令，便可让容器自动加入 Weave 网络。它对发送至 Docker 守护进程的 `docker run` 请求进行拦截，让 Weave 有机会改变网络配置和更改启动容器的请求，从而使容器改用 Weave 的网络栈。这些步骤完成之后，更改后的启动请求便会转发到 Docker 守护进程。这个拦截技术通过 `eval $(weave proxy-env)` 命令实现，它把 `DOCKER_HOST` 环境变量改为指向 Weave 代理，而不是真正的 Docker 守护进程。

Weave 还会创建一个 weave 网桥，你可以通过 `ifconfig` 命令看到它。每个容器，包括 Weave 路由器，都经由一对 veth 接口连接到这个网桥。

Weave 可以把容器放在不同的子网，以隔离不同的应用程序。它还支持信息加密，使 Weave 网络可用于不可信任的网络连接。有关 Weave 的架构和功能的完整介绍，请参阅官方文档（https://www.weave.works/docs/net/latest/introducing-weave/）。

Weave 的重点是要建立卓越的开发者体验，让开发者无需大费周章便能实现容器互联和跨主机的服务发现。

> **以插件方式运行 Weave**
>
> Weave 还可以通过 Docker 的插件框架运行，从而取代代理容器。
> 在撰写这本书的时候，插件方式还有一些限制，主要是由于联网插件的 API 仍在开发中，无法为 Weave 和其他联网插件提供所需的信息以及 hook。例如，当集群重启后，目前 Weave 的配置信息是会丢失的。考虑到这一点，上面给出的操作会比较可取。
>
> 不过当你读到这里的时候，这些问题很可能已经得到解决。

11.5.3　Flannel

Flannel（https://github.com/coreos/flannel）是来自 CoreOS 的一个跨主机联网解决方案。它主要用于以 CoreOS 为基础的集群，不过也完全可以用在其他的栈上。

Flannel 为每一台主机分配一个子网，然后按子网给容器分配 IP 地址。Flannel 与 Kubernetes（参见 12.1.3 节）搭配使用时效果很好，它可以为每个 pod 分配唯一且可经路由传送的 IP。Flannel 会在每台主机上运行一个守护进程，并从 etcd 取得配置（参见 11.2.1 节），因此集群必须已经配置好来使用 etcd。它有一系列的后端可供选择，列举如下。

udp

　　它是默认的后端，把第 2 层的网络信息封装成 UDP 包，在现有的网络上传送时形成一个叠加的网络。

vxlan

　　使用 VXLAN 封装网络数据包。由于这个工作是在内核中完成，它应该比 UDP 快很多，UDP 需要经过用户空间。

aws-vpc

　　用于在 Amazon EC2 上设置网络。

host-gw

　　使用远程 IP 地址设置连接子网的 IP 路由。要求主机之间在第 2 层网络已有连接。

gce

　　用于在 Google Compute Engine 上设置网络。

与之前一样，我们用 Docker Machine 来学习如何使用 Flannel，但这次有点复杂，因为 Flannel 守护进程需要在 Docker 引擎运行起来之前就已经配置好 `flannel0` 网桥，这使得我们不太容易以容器方式运行 Flannel。为了解决这个问题，可以采用一些不太一般的引导方式，例如通过第二个 Docker 守护进程执行 Flannel，不过我认为直接在主机上运行一个 flannel 的进程会更简单。此外，Flannel 还依赖于 etcd，我们也会以单独进程的方式运行 etcd。

坦白来讲，这个用例不太适合使用由 Docker Machine 配置的虚拟机，因为它涉及很多配置步骤，其中还需要撤销一些 Docker Machine 已做好的配置。然而，这个练习对于 Flannel 的入门还是很有启发性的，它应该能够帮助你在自己的网络设施上使用 Flannel。

Flannel 和 etcd 版本

这些范例中使用的 etcd 和 Flannel 版本分别为 2.0.13 和 0.5.1。由于 Flannel 的开发正进行得如火如荼，如果你使用的版本较新，当你发现本书所讲的和你的情况有所不同时，请不要感到诧异。

这个范例中将设置两台主机，分别为 `flannel-1` 和 `flannel-2`，并确保当中的容器能够互通。首先来部署两个 VirtualBox 虚拟机作为我们的主机：

```
$ docker-machine create -d virtualbox flannel-1
...
$ docker-machine create -d virtualbox flannel-2
...
$ docker-machine ip flannel-1 flannel-2
192.168.99.102
192.168.99.103
```

你要记住每台机器的 IP 地址，因为设置 etcd 时需要用到。现在我们在 `flannel-1` 上安装 etcd，但必须先停止 Docker 守护进程并删除 `docker0` 网桥：

```
$ docker-machine ssh flannel-1
...
docker@flannel-1:~$ sudo /usr/local/etc/init.d/docker stop
docker@flannel-1:~$ sudo ip link delete docker0
```

现在下载并解压 etcd：

```
docker@flannel-1:~$ curl -sL https://github.com/coreos/etcd/releases/download/\
v2.0.13/etcd-v2.0.13-linux-amd64.tar.gz -o etcd.tar.gz
docker@flannel-1:~$ tar xzvf etcd.tar.gz
```

然后启动 etcd，这里使用了一些环境变量，使命令读起来比较容易：

```
docker@flannel-1:~$ HOSTA=192.168.99.102
docker@flannel-1:~$ HOSTB=192.168.99.103
docker@flannel-1:~$ nohup etcd-v2.0.13-linux-amd64/etcd \
  -name etcd-1 -initial-advertise-peer-urls http://$HOSTA:2380 \
  -listen-peer-urls http://$HOSTA:2380 \
  -listen-client-urls http://$HOSTA:2379,http://127.0.0.1:2379 \
  -advertise-client-urls http://$HOSTA:2379 \
```

```
  -initial-cluster-token etcd-cluster-1 \
  -initial-cluster \
etcd-1=http://$HOSTA:2380,etcd-2=http://$HOSTB:2380 \
  -initial-cluster-state new &
```

nohup 命令让 etcd 在我们登出主机后还能继续运行。所有输出将会被记录在 nohup.out 文件中。

现在 flannel-1 上的 etcd 配置工作已经完成。当 flannel-2 的配置工作也完成后，我们就会回到 flannel-1 安装 Flannel。

```
docker@flannel-1:~$ exit
$ docker-machine ssh flannel-2
docker@flannel-2:~$ sudo /usr/local/etc/init.d/docker stop
docker@flannel-2:~$ sudo ip link delete docker0
docker@flannel-2:~$ curl -sL https://github.com/coreos/etcd/releases/\
download/v2.0.13/etcd-v2.0.13-linux-amd64.tar.gz -o etcd.tar.gz
docker@flannel-2:~$ tar xzvf etcd.tar.gz
```

启动 etcd 的方法和之前一样，除了 IP 地址被互换了：

```
docker@flannel-2:~$ HOSTA=192.168.99.102
docker@flannel-2:~$ HOSTB=192.168.99.103
docker@flannel-2:~$ nohup etcd-v2.0.13-linux-amd64/etcd \
  -name etcd-2 -initial-advertise-peer-urls http://$HOSTB:2380 \
  -listen-peer-urls http://$HOSTB:2380 \
  -listen-client-urls http://$HOSTB:2379,http://127.0.0.1:2379 \
  -advertise-client-urls http://$HOSTB:2379 \
  -initial-cluster-token etcd-cluster-1 \
  -initial-cluster \
etcd-1=http://$HOSTA:2380,etcd-2=http://$HOSTB:2380 \
  -initial-cluster-state new &
```

现在来下载 Flannel 吧：

```
docker@flannel-2:~$ curl -sL https://github.com/coreos/flannel/releases/\
download/v0.5.1/flannel-0.5.1-linux-amd64.tar.gz -o flannel.tar.gz
docker@flannel-2:~$ tar xzvf flannel.tar.gz
```

接下来需要在 etcd 中添加一些配置来告诉 Flannel 可用的 IP 范围：

```
docker@flannel-2:~$ ./etcd-v2.0.13-linux-amd64/etcdctl
                    set /coreos.com/network/config '{ "Network": "10.1.0.0/16" }'
```

现在就可以启动 Flannel 的守护进程了。请注意，我们需要告诉 Flannel 使用 eth1 接口，因为这个接口可以与其他虚拟机通信：

```
docker@flannel-2:~$ nohup sudo ./flannel-0.5.1/flanneld -iface=eth1 &
```

Flannel 已经运行起来了，执行 ifconfig 可以看到 Flannel 的网桥：

```
docker@flannel-2:~$ ifconfig flannel0
flannel0  Link encap:UNSPEC  HWaddr 00-00-00-00-00-00-00-00-00-00-00-00-00-00-...
          inet addr:10.1.37.0  P-t-P:10.1.37.0  Mask:255.255.0.0
          UP POINTOPOINT RUNNING NOARP MULTICAST MTU:1472 Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
```

```
        TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
        collisions:0 txqueuelen:500
        RX bytes:216 (216.0 B)  TX bytes:221 (221.0 B)
```

请注意，它被分配的地址位于我们给 Flannel 配置的可用 IP 范围中。

下一步要做的就是配置 Docker，使其能够使用 Flannel。如果你使用不同的虚拟机镜像或者是裸机，那么可以使用与 Flannel 一起分发的 mk-docker-opts.sh 脚本来自动配置 Docker。但由于我们的 VirtualBox 镜像中没有 bash，这些需要我们来手动完成。首先来看一下 Flannel 为我们创建的 /run/flannel/subnet.env 文件：

```
docker@flannel-2:~$ cat /run/flannel/subnet.env
FLANNEL_SUBNET=10.1.79.1/24
FLANNEL_MTU=1472
FLANNEL_IPMASQ=false
```

我们需要把 Docker 守护进程的 `--bip` 参数设置为 `FLANNEL_SUBNET` 的值，同时把 `--mtu` 参数设置为 `FLANNEL_MTU` 的值，这样做是为了告诉 Docker 使用与 Flannel 兼容的 IP 地址和 MTU[14]。Docker 守护进程的参数是在 /var/lib/boot2docker/profile 文件中配置的。修改后的文件如下（在虚拟机中可以通过 `sudo vi` 命令进行修改）：

```
docker@flannel-2:~$ cat /var/lib/boot2docker/profile

EXTRA_ARGS='
--label provider=virtualbox
--bip 10.1.79.1/24
--mtu 1472
'
CACERT=/var/lib/boot2docker/ca.pem
DOCKER_HOST='-H tcp://0.0.0.0:2376'
DOCKER_STORAGE=aufs
DOCKER_TLS=auto
SERVERKEY=/var/lib/boot2docker/server-key.pem
SERVERCERT=/var/lib/boot2docker/server.pem
```

现在可以重启 Docker 引擎：

```
docker@flannel-2:~$ sudo /etc/init.d/docker start
hostname: flannel-2: Unknown host
Need TLS certs for flannel-2,,10.0.2.15,192.168.99.103
docker@flannel-2:~$ exit
```

最后，我们需要在 `flannel-1` 上重复这些步骤：

```
$ docker-machine ssh flannel-1
...
docker@flannel-1:~$ curl -sL https://github.com/coreos/flannel/releases/\
download/v0.5.1/flannel-0.5.1-linux-amd64.tar.gz -o flannel.tar.gz
v0.5.1/flannel-0.5.1-linux-amd64.tar.gz -o flannel.tar.gz
docker@flannel-1:~$ tar xzvf flannel.tar.gz
...
docker@flannel-1:~$ nohup sudo ./flannel-0.5.1/flanneld -iface=eth1 &
```

注 14：MTU 是指最大传输单元（Maximum Transmission Unit），控制网络中允许的数据包大小。

```
docker@flannel-1:~$ cat /run/flannel/subnet.env
FLANNEL_SUBNET=10.1.83.1/24  ❶
FLANNEL_MTU=1472
FLANNEL_IPMASQ=false
docker@flannel-1:~$ sudo vi /var/lib/boot2docker/profile
...
docker@flannel-1:~$ sudo /etc/init.d/docker start
hostname: flannel-1: Unknown host
Need TLS certs for flannel-1,,10.0.2.15,192.168.99.102
docker@flannel-1:~$ exit
```

❶ 注意，这个值必须与 `flannel-2` 不同，这样两个主机才能为容器分配不同范围的 IP 地址。

现在，一切已准备就绪，让我们确认一下容器之间是否能够互通。首先在 `flannel-1` 上用 Netcat 工具监听网络端口：

```
$ eval $(docker-machine env flannel-1)
$ docker run --name nc-test -d amouat/network-utils nc -l 5001
...
```

网络工具容器

当网络出现问题时，为了能够进行各种网络测试，拥有一个包含各种网络工具的镜像就非常方便了。为此，我已经构建好一个名为 `amouat/network-utils` 的镜像。它包含 curl、Netcat、traceroute、dnsutils 等工具，甚至还有 jq，方便将 REST API 的 JSON 输出排列整齐。

这是其中一个用法示范：

```
$ docker run -it amouat/network-utils
root@7e80c9731ea0:/# curl -s https://api.github.com\
/repos/amouat/network-utils-container\
      | jq '. .description'
"Docker container with some network utilities"
```

然后找出 Flannel 分配给容器的 IP 地址：

```
$ IP=$(docker inspect -f {{.NetworkSettings.IPAddress}} nc-test)
$ echo $IP
10.1.83.2
```

你会发现这个 IP 地址是在我们之前设置的地址范围内。现在，让我们在 `flannel-2` 上启动 Netcat 并尝试连接至 `nc-test` 容器：

```
$ eval $(docker-machine env flannel-2)
$ docker run -e IP=$IP \
    amouat/network-utils sh -c 'echo -n "hello" | nc -v $IP 5001'
Unable to find image 'amouat/network-utils:latest' locally
...
Status: Downloaded newer image for amouat/network-utils:latest
Connection to 10.1.83.2 5001 port [tcp/*] succeeded!
```

如果打开 `nc-test` 容器的日志，那么可以看到刚才发送的消息：

```
$ eval $(docker-machine env flannel-1)
$ docker logs nc-test
hello
```

关于 Flannel 的介绍就到此为止。我们实现了两个分别使用自己的 IP 地址的容器，它们在不同的主机上进行通信。虽然好像花了很多工夫才做到这一点，但请记住，未来这些操作都有望实现自动化，当新主机加入集群时就无需再重复这些步骤，而且那时使用 CoreOS 栈的用户应能享受到开箱即用的体验。

然而现在 identidock 还没有运行起来。虽然我们实现了跨主机的联网，但还没有服务发现，这需要使用之前介绍过的方案，例如 SkyDNS，或重写 identidock 以便使用 etcd。

11.5.4　Calico项目

Calico 项目（以下简称 Calico）的联网方式稍有不同，为便于理解，这里从 OSI 模型（https://en.wikipedia.org/wiki/OSI_model）的角度来解释。概念上来讲，OSI 模型把网络分成 7 个层。大多数的联网方案，例如 Weave 和 Flannel（当使用 UDP 后端时），都是把第 2 层[15] 数据封装，形成一个在现有的网络上叠加的网络。而 Calico 使用标准的 IP 路由和网络工具，提供一个第 3 层[16] 的方案。

纯粹的第 3 层解决方案的主要优点是简单和高效。Calico 的主要操作模式不涉及封装，专为有能力控制物理网络架构的机构在数据中心内使用。Calico 网络的路由使用边界网关协议（Border Gateway Protocol，BGP）建立，BGP 是一个历史悠久的网络协议，它支撑着整个互联网，在这里它被用于连接数据中心网络内部和边缘的路由器。这种方法使 Calico 能够用在各种各样的第 2 层和第 3 层的物理拓扑之上。对外连接也无需使用 NAT，只要安全政策允许和 IP 网络可用，容器就能直接连至公共 IP。

Calico 的缺点是它的主要模式不能在公有云上使用，因为用户无法控制它的网络架构。不过，Calico 仍然可以在公有云上使用，但通常需要 IP-in-IP 隧道来提供连接。

另外值得一提的是 Calico 的安全模型，它对容器之间能否互相通信提供了细粒度的控制。

注意，这里使用的是试验版

范例中的虚拟机使用的 Docker 为试验版，因此它与你使用的版本必定存在些许差异。当你正阅读本书的时候，Docker 的稳定版已经能够支持网络插件，你应该使用它。

为了完整起见，接下来的范例使用的都是由以下命令部署的 Digital Ocean 虚拟机：

```
$ docker-machine create -d digitalocean \
        --digitalocean-access-token=<token> \
        --digitalocean-private-networking \
        --engine-install-url \
```

注 15：OSI 模型中的“数据链路层”（data link layer），MAC 地址属于这一层。

注 16：OSI 模型中的“网络层”（network layer），IPv4 和 IPv6 属于这一层。

```
            "https://experimental.docker.com" calico-1
...
$ docker-machine create -d digitalocean \
          --digitalocean-access-token=<token> \
          --digitalocean-private-networking \
          --engine-install-url \
            "https://experimental.docker.com" calico-2
...
$ docker-machine ssh calico-1
root@calico-1:~# docker -v
Docker version 1.8.0-dev, build 3ee15ac, experimental
```

我还在其中一台主机上手动安装了 Consul，Docker 守护进程需要它来共享网络配置信息。

当你读到这里的时候，很多东西将已改变，你要做好心理准备，这里的范例需要少许改动才能工作。不要试图使用和我一样的 Calico 和 Docker 版本，你应该使用当前最新且仍在维护的稳定版。

下面假设有两个由 Docker Machine 配置的虚拟机，分别称为 calico-1 和 calico-2，它们可以在 <calico-1 ipv4> 和 <calico-2 ipv4> 地址上互相通信（地址可以是内网的，也不必能够从互联网访问得到）。这里我使用的是 Digital Ocean 的云服务，但其他云服务也应该都差不多。

首先需要做的就是在每台主机上配置 etcd，因为 Calico 采用 etcd 作为主机之间共享网络信息的工具：

```
$ HOSTA=<calico-1 ipv4>
$ HOSTB=<calico-2 ipv4>
$ eval $(docker-machine env calico-1)
$ docker run -d -p 2379:2379 -p 2380:2380 -p 4001:4001 \
  --name etcd quay.io/coreos/etcd \
  -name etcd-1 -initial-advertise-peer-urls http://${HOSTA}:2380 \
  -listen-peer-urls http://0.0.0.0:2380 \
  -listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
  -advertise-client-urls http://${HOSTA}:2379 \
  -initial-cluster-token etcd-cluster-1 \
  -initial-cluster \
etcd-1=http://${HOSTA}:2380,etcd-2=http://${HOSTB}:2380 \
  -initial-cluster-state new
...
b9a6b79e42a1d24837090de4805bea86571b75a9375b3cf2100115e49845e6f3
$ eval $(docker-machine env calico-2)
$ docker run -d -p 2379:2379 -p 2380:2380 -p 4001:4001 \
  --name etcd quay.io/coreos/etcd \
  -name etcd-2 -initial-advertise-peer-urls http://${HOSTB}:2380 \
  -listen-peer-urls http://0.0.0.0:2380 \
  -listen-client-urls http://0.0.0.0:2379,http://0.0.0.0:4001 \
  -advertise-client-urls http://${HOSTB}:2379 \
  -initial-cluster-token etcd-cluster-1 \
  -initial-cluster \
etcd-1=http://${HOSTA}:2380,etcd-2=http://${HOSTB}:2380 \
```

```
  -initial-cluster-state new
...
2aa2d8fee10aec4284b9b85a579d96ae92ba0f1e210fb36da2249f31e556a65e
```

现在 etcd 已经启动，可以开始安装 Calico。当你读到这里的时候，这些步骤可能已经有点过时，但大致应该差不多。首先是下载 Calico：

```
$ docker-machine ssh calico-1
...
root@calico-1:~# curl -sSL -o calicoctl \
  https://github.com/Metaswitch/calico-docker/releases/download/v0.5.2/calicoctl
root@calico-1:~# chmod +x calicoctl
```

Calico 需要一些由 `xt_set` 内核模块提供的 IP tables 功能，因此需要加载它：

```
root@calico-1:~# modprobe xt_set
```

我们需要告诉 Calico 可分配的 IP 地址范围，这可以通过 `pool add` 子命令来完成：

```
root@calico-1:~# sudo ./calicoctl pool add 192.168.0.0/16 --ipip --nat-outgoing
```

其中的 `--ipip` 参数告诉 Calico，主机之间需要建立 IP-in-IP 隧道，当主机之间没有直接的第 2 层连接时，便需要使用这个参数。这个命令只需在一台主机上运行。

下一步便是启动 Calico 服务，其中包括 Docker 的联网插件，它将以容器运行：

```
root@calico-1:~# sudo ./calicoctl node --ip=<calico-1 ipv4>
WARNING: ipv6 forwarding is not enabled.
Pulling Docker image calico/node:v0.5.1
Calico node is running with id: d72f2eb6f10ea24a76d606e3ee75bf...
```

现在以同样方式配置另一台主机：

```
$ docker-machine ssh calico-2
...
root@calico-2:~# curl -sSL -o calicoctl
https://github.com/Metaswitch/calico-docker\
/releases/download/v0.5.2/calicoctl
root@calico-2:~# chmod +x calicoctl
root@calico-2:~# modprobe xt_set
root@calico-2:~# sudo ./calicoctl node --ip=<calico-2 ipv4>
WARNING: ipv6 forwarding is not enabled.
Pulling Docker image calico/node:v0.5.1
Calico node is running with id: b880fac45feb7ebf3393ad4ce63011a2...
root@calico-2:~#
```

现在可以再一次启动 identidock 程序。首先在 `calico-2` 上启动 Redis，执行命令时需要加上 `--publish-service redis.anet.calico` 参数，它的作用是创建一个全新的 Calico 网络，名叫“anet”，以及一个“redis”服务：

```
root@calico-2:~# docker run --name redis -d \
       --publish-service redis.anet.calico redis:3
....
6f0db3fe01508c0d2fc85365db8d3dcdf93edcdaae1bcb146d34ab1a3f87b22f
```

如果现在登录到 calico-1，你会发现可以连接到相同的网络，而且能够访问 Redis 容器：

```
root@calico-2:~# exit
$ docker-machine ssh calico-1
root@calico-1:~# docker run --name redis-client \
        --publish-service redis-client.anet.calico \
        redis:3 redis-cli -h redis ping
...
PONG
```

现在让我们在同一网络上启动 dnmonster 和 identidock 容器：

```
root@calico-1:~# docker run --name dnmonster \
        --publish-service dnmonster.anet.calico -d amouat/dnmonster:1.0
...
fba8f7885a2e1700bc0e263cc10b7d812e926ca7447e98d9477a08b253cafe0
root@calico-1:~# docker run --name identidock \
        --publish-service identidock.anet.calico -d amouat/identidock:1.0
...
589f6b6b17266e59876dfc34e15850b29f555250a05909a95ed5ea73c4ee7115
```

是时候测试一下我们的应用了：

```
root@calico-1:~# docker exec identidock curl -s localhost:9090
<html><head><title>Hello...
```

太棒了，我们已经成功利用 Calico 把 identidock 跑起来了。刚才我们只是在容器内访问 identidock，因为客户端必须也在 Calico 的网络内。当然，使应用能够从其他网络访问是完全可行的，譬如互联网，但这样做需要更多配置，而这部分在未来有可能会改变，因此暂且略过。

Calico 的实现有赖于它背后的一些组件。

etcd

　用于存储和发布主机以及容器的信息。

BIRD

　BIRD 互联网路由守护进程（The BIRD Internet Routing Daemon）[17] 利用 BGP 实现主机和容器之间的 IP 通信路由。

Felix

　它是一个运行在每台计算主机上的 Calico 代理程序，利用来自 etcd 的数据配置本地网络策略。

Calico 插件

　当 Docker 容器创建时，负责建立网络连接，并把容器记录在 etcd 中。

注 17：据我所知，BIRD 中的 B 并不代表任何词，它只是以一种令人讨厌的递归方式代表 BIRD 这个词。

目前，Calico 把重点放在机构能够控制网络架构的使用场景，例如大型企业的私有云，为虚拟机及容器提供高效和相对简单的联网技术。同时，Calico 插件为使公有云上的容器能够互联而提供类似的解决方案。

11.6 总结

服务发现在现今的分布式和动态系统中往往是必不可少的功能。容器和服务都会不断变化，它们会因需求或故障而被停止、启动和迁移。考虑到这些情况，需要手动更改路由的解决方案是行不通的。

我们接触过的大多数服务发现方案都能支持基于 DNS 的查找操作，客户端只需以名字称呼服务，系统便负责把客户端连接到相应的服务实例。对于客户端以及现有的应用程序和工具而言，支持这种做法非常简单，但在高度动态的系统中，DNS 则可能是个负担。DNS 的响应一般会被缓存，当服务在主机之间迁移时，将会导致延迟和错误发生。负载均衡一般充其量使用循环的方式，大部分情况下并不是一个理想的选择。此外，客户端可能希望能够以自己的逻辑在可用的服务中选择合适的，如果系统能够提供丰富的 API，实现这个功能就简单多了。

使用哪一种服务发现工具较为合适，在很大程度上取决于你的用例。对于大多数项目而言，使用的相关工具依项目的软件需求或已使用的平台而定（譬如，因 Mesos 而使用 ZooKeeper，因 GKE 而使用 etcd）。如果你属于这种情况，使用已有的工具比较合适，最好不要再把其他工具拉进来。在 etcd（或 etcd 加上 SkyDNS）和 Consul 之间作出选择要困难得多。两者都属于比较新的项目（etcd 时间稍微长一点），但底层都使用了可靠的算法。Consul 默认包含 DNS 支持和一些先进的功能，这往往使它成为更好的选择。如果把 etcd 与 Consul 进行比较，etcd 可以说不会强迫你去按照它的那一套行事，而且它的键值存储更先进，因此在需要大量定制的场景下，它可能是一个更好的选择。如果你的程序使用 API 来寻找服务，或者你使用的联网解决方案已经能够提供服务名称解析，那么 Consul 和 SkyDNS 提供的 DNS 支持对你来说可能就不太重要了。

相对来说，选择一个合适的联网解决方案更困难，主要原因是这方面的技术还不太成熟。在未来的数个月内，将会有更多的解决方案面世（尤其是以联网插件的形式），而它们之间的差异化应该会更明显。到目前为止，我还没有对任何解决方案进行性能和扩展性测试，因为我预计随着供应商对程序的优化，以及针对特定用例作出的改良，将来的测试结果将会大大不同。话虽如此，从当前的解决方案来看，我持下列观点。

- Docker 的 Overlay 联网方案很有可能成为开发过程中最常用的，只因它是 Docker 默认自带的“内附电池”方案。观乎它未来的稳定性和效率，它可能也适用于云端上的小规模部署。
- Weave 的重点集中于易用性和开发者体验，这使它成为开发过程中另一个不错的选择。Weave 还包括加密和穿透防火墙等功能，使它在诸如跨云端的部署的情况下非常有吸引力。
- Flannel 被应用于 CoreOS 的栈中，它针对不同场景提供了专门的后端。撰写本书的时候，在开发环境中使用 Flannel 可能会大费周章（随着插件的开发，这个情况应有所改变），但在一些生产环境中，它能提供高效并简单的解决方案。

- Calico 项目主要服务于那些能够控制自己的网络结构的大型机构或数据中心。在这些场景中，Calico 项目基于第 3 层网络实现的技术，可以提供一个既简单又高效的解决方案。不仅如此，Calico 项目的网络插件看起来不仅易用，而且速度还不错，这使它在开发环境和单一云平台的部署中具有非凡的吸引力。
- 你可以选择建立一套自己的方案。运维人员有时候清楚地知道，为了提高效率，各部件应如何连接起来。这种情况下，你可以创建自己的联网插件，或使用 pipework 这种工具加入特别的底层逻辑。你还可以使用容器或主机联网模式，因为它们没有 Docker 网桥和 NAT 规则带来的开销，不过所有容器必须共享 IP 地址。

究竟什么是正确的选择，很大程度取决于你的特定需求和你所使用的平台。你可能会发现，某些解决方案的运行速度比其他的更快或更慢，或者能够实现别的方案实现不了的用例。没有任何方法比实际测试更管用，我建议把你的应用程序的运行场景复制至测试环境，然后尝试应用不同的解决方案，以找出最合适的一个。

第12章

编排、集群和管理

大多数软件系统都会随着时间演变。新的功能会添加，旧的功能会删减。不断改变的用户需求，意味着一个高效的系统必须能够把资源快速向上和向下扩展。近乎零停机时间的要求，意味着发生故障时需要能够自动切换到已预先部署的备份系统，而系统通常位于其他的数据中心或地区。

除此之外，机构经常运行着多个这样的系统，或需要执行一些不经常运行并与主系统分开的任务，例如数据挖掘，而这些任务需要相当多的资源，或需要与现有的系统通信。

当使用多个资源时，我们需要确保资源得到有效利用而不是被闲置，同时仍然能够应付瞬间的需求高峰，这是非常重要的。在成本效益与快速扩展能力之间取得平衡是一项艰难的任务，所幸的是解决方法有很多种。

上述这一切意味着，即使运行一个稍微复杂一点的系统，也已经涉及很多管理任务并充满了挑战，其中的复杂度不容小觑。很快你就会发现，为每一个机器单独进行管理是不现实的，与其逐一给机器打补丁和更新，不如对它们进行统一管理。当一台机器出了问题时，你应该销毁和更换它，而不是“调养”并使它恢复正常。[1]

现今已有各式各样的软件工具和解决方案来协助我们面对这些挑战，它们大致涵盖以下方面。

集群（Clustering）

　　把“主机”组合并通过网络连接起来，虚拟机或裸机皆可。集群看起来应该像一个单一资源，而不是一组互不相干的机器。

注 1：这两种思路迥异的做法，通常被比喻为“宠物与牲口”（pets versus cattle）。

编排（Orchestration）

协调各组件使它们共同运作。在适合的主机上启动容器并把它们连接起来。编排系统也可能包括扩展的支持、自动故障切换，以及节点的负载均衡。

管理（Management）

监督系统及支持各种管理任务。

本章首先会介绍 Docker 生态系统中主要的编排和集群工具：Swarm、fleet、Kubernetes 和 Mesos。Swarm 是 Docker 自带的集群解决方案，很大程度上还能解决编排方面的难题，特别是与 Docker Compose 一起使用时。Fleet 是 CoreOS 使用的一个底层集群与调度系统。Kubernetes 是一个层次更高的编排方案，它的设计要求你按照它的规则行事，默认支持故障切换和扩展功能，并可以在其他集群方案之上运行。Mesos 是一个底层的集群解决方案，它与更高层次的“框架”搭配使用，以提供一个稳健和完整的集群和编排解决方案。

之后还会介绍几个“容器管理平台”，包括 Rancher、Clocker 和 Tutum，并为跨主机的容器系统提供了管理界面（包括 GUI 和 CLI）。这些平台一般利用之前介绍过的基础组件，例如 Overlay 联网方案，不过这些平台会把不同组件整合成一个集成的产品。

这一章的代码可以在本书的 GitHub（https://github.com/using-docker/orchestration）上找到。

可以通过以下命令获取这一章的代码：

```
$ git clone -b \
  https://github.com/using-docker/orchestration/
...
```

除此之外，也可以从 GitHub 的项目页面直接下载代码。

12.1 集群和编排工具

本节将研究可用于 Docker 的主要集群和编排工具，包括 Swarm、fleet、Kubernetes 和 Mesos。本节将会介绍每一个工具的独特功能，以及如何将其运用到我们的 identidock 范例中。

12.1.1 Swarm

Swarm（https://docs.docker.com/swarm/）是 Docker 自带的集群工具。Swarm 使用标准的 Docker API，也就是说，容器可以通过正常的 `docker run` 命令启动，Swarm 会在运行容器时负责选择合适的主机。这也意味着，其他使用 Docker API 的工具和定制的脚本，无需任何修改便能使用 Swarm，享受从单一主机架构转变为集群的便利。

Swarm 的基本架构相当简单：每台主机上运行一个 Swarm 的*代理*（agent），以及在一台主机上运行一个 Swarm 的*主管*（manager，对于小规模的测试集群，这台主机也可以同时运行代理）。主管负责所有主机上容器的编排和调度。Swarm 能够以高可用性模式运行，通过利用 etcd、Consul 或 ZooKeeper，使得故障发生时能够切换到备份的主管主机。Swarm

把寻找主机和将主机加入集群称为*发现*（discovery），这个功能有好几种实现方法，其中默认使用的是基于*令牌*（token）的方法，它会把主机的地址保存在 Docker Hub 的一个列表上。

本书写作时，Swarm 的版本是 0.4，仍在开发中。值得注意的是，它没有跨主机的联网功能，因此任何已连接的容器必须在同一主机上运行。不过这个问题很可能在你读到这里的时候已经解决了；跨主机联网的功能将通过与仍处于开发阶段的联网插件集成来实现。

为了能快速上手 Swarm，让我们来建立一个小型的虚拟机集群。下面将利用 Docker Machine 创建虚拟机，并通过默认的令牌发现方法把它们连接起来。首先需要执行 `swarm create` 命令，为集群创建一个令牌：

```
$ SWARM_TOKEN=$(docker run swarm create)
$ echo $SWARM_TOKEN
26a4af8d51e1cf2ea64dd625ba51a4ff
```

现在可以创建主管（或叫作 master）的主机：

```
$ docker-machine create -d virtualbox \
        --engine-label dc=a \
        --swarm --swarm-master \
        --swarm-discovery token://$SWARM_TOKEN \
        swarm-master
Creating VirtualBox VM...
Creating SSH key...
Starting VirtualBox VM...
Starting VM...
To see how to connect Docker to this machine, run: docker-machine env swarm-ma...
```

Docker Machine 创建了一个新的 VirtualBox 虚拟机，名为 `swarm-master`，并以之前生成的令牌加入 Swarm 集群。我们还给主机上的 Docker 引擎附上 `dc=a` 标签，之后会解释这样做的目的。接着要创建另外两个虚拟机来组成我们的集群，名字分别为 `swarm-1` 和 `swarm-2`：

```
$ docker-machine create -d virtualbox \
    --engine-label dc=a \
    --swarm \
    --swarm-discovery token://$SWARM_TOKEN \
    swarm-1
...
$ docker-machine create -d virtualbox \
    --engine-label dc=b \
    --swarm \
    --swarm-discovery token://$SWARM_TOKEN \
    swarm-2
...
```

注意，`swarm-1` 的标签是 `dc=a`，而 `swarm-2` 的标签是 `dc=b`。

我们可以通过 Docker Hub 的 API，手动验证这些节点是否已被加入集群：

```
$ curl https://discovery-stage.hub.docker.com/v1/clusters/$SWARM_TOKEN
["192.168.99.103:2376","192.168.99.102:2376","192.168.99.101:2376",
"192.168.99.100:2376"]
```

命令中列出的 IP 地址是我们创建的虚拟机地址，只需 Swarm 主管能够访问便可。也可以通过执行 `swarm list` 命令获得同样的信息（适用于任何服务发现方法）。你可以下载 Swarm 的二进制程序来执行它，但最简单的方法其实还是使用 Swarm 镜像，如同我们在创建令牌时所做的：

```
$ docker run swarm list token://$SWARM_TOKEN
192.168.99.108:2376
192.168.99.109:2376
192.168.99.107:2376
```

图 12-1 展示了我们创建的一个简单集群。标签 `dc=a` 和 `dc=b` 分别表示数据中心 A 和数据中心 B。虽然把两个在笔记本电脑上运行的虚拟机当作数据中心，就如同把蚊子当作喷气式飞机，但作为例子已经绰绰有余。你还可以利用类似的命令，把强大的云资源很轻松地添加到你的集群中。

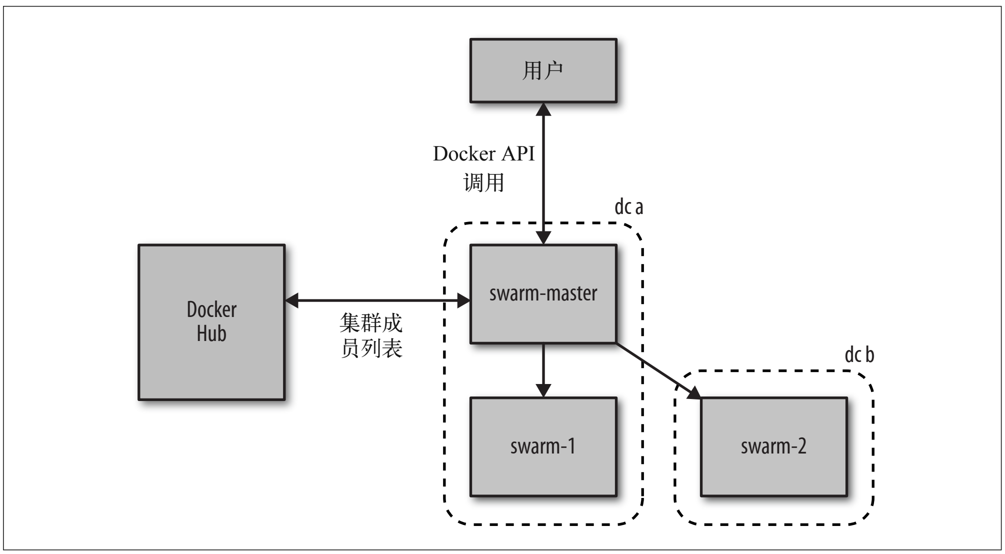


图 12-1：Swarm 集群示例

> **Swarm 的服务发现功能**
>
> 默认的基于令牌的发现功能对于快速上手非常有用，但却有一个明显的缺点，那就是要求所有主机必须能够访问 Docker Hub，这是个单点故障的潜在风险。
>
> 另外还有其他可用的发现机制，可以只是简单地向 Docker 管理程序提供 IP 地址列表，也可以使用分布式存储，如 etcd、Consul 和 ZooKeeper 等。
>
> 服务发现方法的完整信息参见文档（https://docs.docker.com/swarm/discovery/）。

现在把 Docker 客户端连接至 Swarm 主管，看看 `docker info` 告诉我们什么：

```
$ eval $(docker-machine env --swarm swarm-master)
$ docker info
Containers: 4
Images: 3
Role: primary
Strategy: spread
Filters: affinity, health, constraint, port, dependency
Nodes: 3
 swarm-1: 192.168.99.102:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.022 GiB
  └ Labels: dc=a, executiondriver=native-0.2, ...
 swarm-2: 192.168.99.103:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.022 GiB
  └ Labels: dc=b, executiondriver=native-0.2, ...
 swarm-master: 192.168.99.101:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.022 GiB
  └ Labels: dc=a, executiondriver=native-0.2, ....
CPUs: 3
Total Memory: 3.065 GiB
```

返回的结果包括一些集群的详细资料，以及我们创建的 3 个主机（Swarm 中称之为“节点”）。每个节点运行着一个连接到集群的 Swarm 代理容器，`swarm-master` 节点运行着一个用于管理集群的 Swarm 主管容器。在生产环境中，不建议把代理和主管运行在同一个节点上（考虑到故障切换的缘故），但作为演示则无伤大雅。

现在就来测试一下我们的集群吧！

```
$ docker run -d debian sleep 10  ❶
ebce5d18121002f35b2666da4dd2dce189ece9573c8ebeba531d85f51fbad8e8
$ docker ps
CONTAINER ID  IMAGE   COMMAND    ... NAMES
ebce5d181210  debian  "sleep 10" ... swarm-1/furious_bell
```

❶ 这个命令将会下载 `debian` 镜像，因此需要花一些时间完成。与 Swarm 集群通信时，你是不会看到下载进度的。

可以看到，容器已经成功创建，而且已自动安排在 `swarm-1` 主机上。虽然它看似平淡无奇，其实它做了很多事情。首先，Swarm 在背后拦截我们的请求，然后对集群进行分析，最后把请求转发至最合适的主机。

1. 过滤器

过滤器（filter）的作用是控制哪些节点可用于运行容器，有几种过滤器是默认使用的。我们可以尝试启动一些 nginx 容器，看看其中一个默认过滤器如何运作：

```
$ docker run -d -p 80:80 nginx
6d571c0acaa926cea7194255617dcd384375c105b0285ef657c911fb59c729ce
$ docker run -d -p 80:80 nginx
```

```
7b1cd5dade7de5bed418d360c03be72d615222b95e5f486d70ce42af5f9e825c
$ docker run -d -p 80:80 nginx
ab542c443c05c40a39450111ece852e9f6422ff4ff31864f84f2e0d0e6697605
$ docker ps
CONTAINER ID  IMAGE ... PORTS                                NAMES
ab542c443c05  nginx     192.168.99.102:80->80/tcp, 443/tcp   swarm-1/mad_eng...
7b1cd5dade7d  nginx     192.168.99.101:80->80/tcp, 443/tcp   swarm-master/co...
6d571c0acaa9  nginx     192.168.99.103:80->80/tcp, 443/tcp   swarm-2/elated_...
```

你会发现，Swarm 已经把每一个 nginx 容器放在了不同的主机上。如果尝试启动第四个容器，将会发生什么？

```
$ docker run -d -p 80:80 nginx
Error response from daemon: unable to find a node with port 80 available
```

其中一个默认过滤器是*端口*（port），当容器要求使用主机上的某个特定端口时，过滤器会安排容器到一个端口未被占用的节点。由于启动第四个容器时，所有主机的 80 端口已被占用，Swarm 拒绝了这个请求。

约束（constraint）过滤器可按键值对挑选可用的节点子集。可以通过先前为主机设置的标签来了解它的运作方式：

```
$ docker run -d -e constraint:dc==b postgres
e4d1b2991158cff1442a869e087236807649fe9f907d7f93fe4ad7dedc66c460
$ docker run -d -e constraint:dc==b postgres
704261c8f3f138cd590103613db6549da75e443d31b7d8e1c645ae58c9ca6784
 docker ps
CONTAINER ID        IMAGE     ...  NAMES
704261c8f3f1        postgres       swarm-2/berserk_yalow
e4d1b2991158        postgres       swarm-2/nostalgic_ptolemy
...
```

两个容器都被安排在 `swarm-2` 上，因为它是唯一的标签为 `dc=b` 的主机。为了证明这一点，我们也可以使用 `constraint:dc==a` 或 `constraint:dc!=b`：

```
$ docker run -d -e constraint:dc==a postgres
62efba99ef9e9f62999bbae8424bd27da3d57735335ebf553daec533256b01ef
$ docker ps
CONTAINER ID        IMAGE     ... NAMES
62efba99ef9e        postgres      swarm-master/dreamy_noyce
704261c8f3f1        postgres      swarm-2/berserk_yalow
e4d1b2991158        postgres      swarm-2/nostalgic_ptolemy
...
```

可以看到，容器被安排在 `swarm-master` 上，因为它的标签是 `dc=a`。

约束过滤器也可用于过滤各种主机信息，例如主机名、存储驱动和操作系统。

这个过滤器还可以安排容器在特定的地域（例如 `constraint:region!=europe`）或特定的硬件上（例如 `constraint:disk==ssd` 或 `constraint:gpu==true`）启动。

其他的过滤器还包括以下几种。

健康状况（health）

只安排容器运行在“健康的”（healthy）主机上。

依赖关系（dependency）

编排容器时会把依赖它的其他容器一同安排（例如，与它共享数据卷的容器，或互相连接的容器，将会被安排在相同的主机上运行）。

关联（affinity）

允许用户定义容器与其他容器或主机之间的“吸引度”。例如，可以指定容器必须与某个已存在的容器共同安排在一台主机上，或指定容器只能在已经下载了某个镜像的主机上运行。

约束与关联的表达式语法

关联过滤器和约束过滤器的表达式可以使用 == 运算符（表示节点必须匹配的值）或 != 运算符（表示节点不能匹配的值）。

其中还可以使用正则表达式和匹配模式。例如：

```
$ docker run -d -e constraint:region==europe* postgres ❶
$ docker run -d -e constraint:node==/swarm-[12]/ postgres ❷
```

❶ 容器将在 region（区域）标签以 europe 开头的主机上运行。
❷ 容器将在名为 swarm-1 或 swarm-2 的主机上运行（但不能是 swarm-master）。

此外，“非强制性的”约束或关联可以在值的前面以 ~ 表示，调度器会尝试满足这个规则，但如果无法满足，它仍然会在无法匹配规则的资源上运行容器，而不是以失败告终。例如：

```
$ docker run -d -e constraint:dc==~a postgres
```

这个命令将首先尝试在标签为 dc=a 的主机上运行容器，但如果该主机已不能使用，它仍然会在其他主机上运行。

2. 策略

假设在过滤条件应用后，得到多台可用的主机，Swarm 将如何为容器选择呢？答案是取决于所选的策略。以下是一些可供使用的策略。

分散（spread）

将容器放置在负载最小的主机上。

集装（binpack）

将容器放置在负载最多且还有可用空间的容器上。

随机（random）

将容器随机安排在任何主机上。

分散策略使容器均匀地分布到所有主机上。这种方法的主要优点是，当主机停机时，受影响的容器数量将降到最少。集装策略将会尽可能利用主机，从而优化机器的使用率。随机策略主要用于调试。

目前来看，Swarm 最适合小型至中型规模，大概是几十至几百台主机的部署。如果希望在更庞大的集群上运行 Swarm，可以考虑将 Swarm 与 Mesos 整合，这样就能利用 Swarm API 在 Mesos 的基础设施上启动容器，而且 Mesos 已被证实能够稳定地扩展至成千上万台主机的规模。

删除虚拟机

这一章中使用 Docker Machine 来创建大量的虚拟机。由于它们占用大量资源，当你完成了一个例子之后，务必把它们停止并且删除。为此，Docker Machine 提供了很简单的命令：

```
$ docker-machine stop swarm-master
$ docker-machine rm swarm-master
Successfully removed swarm-master
```

Swarm 的主要优点是，它只使用直接的 Docker API，这就意味着你可以把大型的工作负载或应用程序迁移到另一个集群，或改为跨集群的方式执行。如果使用 Mesos 或 Kubernetes，会导致你的架构将难以移植。

12.1.2 fleet

fleet（https://coreos.com/fleet/）是出自 CoreOS 的集群管理工具。它标榜自己为一个“底层的集群引擎”，也就是说，它希望成为一个为上层应用（如 Kubernetes）提供服务的“基础层”。

fleet 最显著的特点是，它的建立是基于 systemd（https://wiki.freedesktop.org/www/Software/systemd/）之上。虽然 systemd 用于给单一机器提供系统和服务的初始化支持，但 fleet 把它延伸至计算机集群。fleet 读取 systemd 的单元（unit）文件，然后把它安排在集群中的一台或多台机器上运行。

fleet 的技术架构如图 12-2 所示。每台机器运行一个*引擎*（engine）和*代理*（agent）。无论何时，集群中只有一个运作中的引擎，但所有代理都在不断运行（为了方便绘图，运作中的引擎没有与机器放在一起，但实际上它运行于其中一台机器上）。systemd 单元文件（以下简称*单元*，unit）会被提交到引擎，引擎会把作业安排在“负载最少”的机器上。单元文件通常只会负责运行容器，而代理则负责启动单元以及报告状态。etcd 被用于辅助建立机器之间的通信，以及保存集群和单元的状态。

这个架构的设计能够支持容错处理；如果一台机器出现任何状况，那么所有被安排在该机器上的单元将在新的主机上重新启动。

fleet 支持各种调度的提示和限制。最基本的层面上，可以把单元安排为*全局*单元，即运行在所有机器上的一个实例，也可以安排它为只在一台机器上运行的单一单元。全局调度非常适合工具类的容器，譬如日志记录和监控等服务。它还支持各种*关联*类型的约束。例如，

可以安排执行健康检查的容器只与应用服务器在一起。也可以把用于调度的元数据与主机关联，让你可以指示容器运行在某一区域的机器上，或运行在已安装某些硬件的机器上。

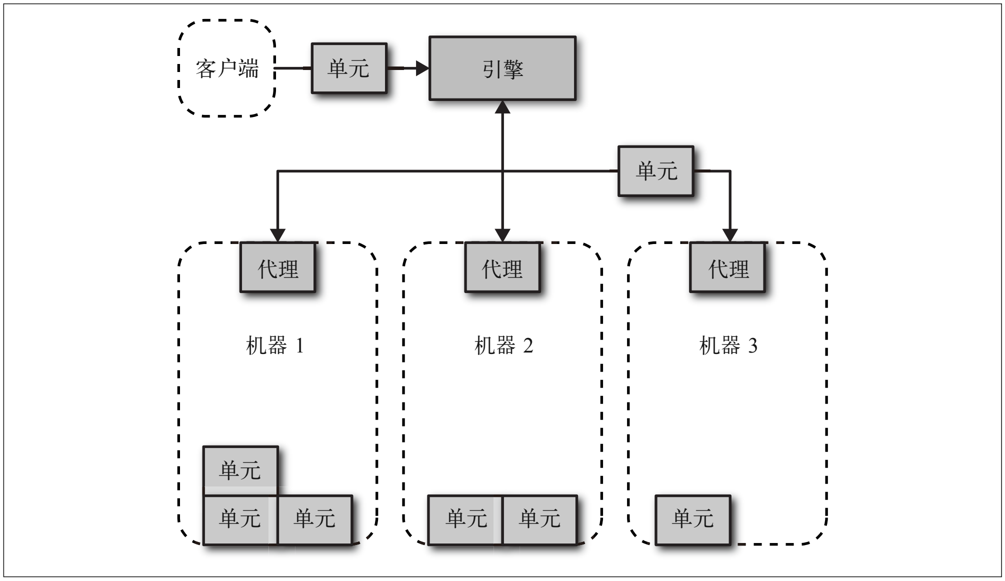


图 12-2：fleet 总体架构

由于 fleet 是基于 systemd 的，它也支持*套接字激活*（socket activation）的概念（即当某个特定端口上有连接发生时，容器便会启动）。这样做的主要好处在于，进程可以在需要时才创建，而无需闲置着等待事情发生。关于套接字的管理，理论上还有其他好处，例如在容器多次重启之间，消息不会丢失。

接下来看一下如何将 identidock 运行在 fleet 的集群上。我在这个范例中创建了一个 GitHub 项目，其中包含了一个 Vagrant 模板，用于启动 3 个虚拟机：

```
$ git clone https://github.com/amouat/fleet-vagrant
...
$ cd fleet-vagrant
$ vagrant up
...
$ vagrant ssh core-01 -- -A
CoreOS alpha (758.1.0)
```

现在已经启动了一个包含 3 个虚拟机的集群，所有虚拟机均使用 CoreOS，并且 Flannel（详见 11.5.3 节）和 fleet 都已安装妥当。可以使用 fleet 的命令行工具 `fleetctl` 获取集群中的机器列表：

```
core@core-01 ~ $ fleetctl list-machines
MACHINE    IP    METADATA
16aacf8b... 172.17.8.103 -
39b02496... 172.17.8.102 -
eb570763... 172.17.8.101 -
```

首先，我们打算使用基于 DNS 的服务发现，因此需要安装 SkyDNS（详见 11.2.2 节）。像这种通用服务，在集群的所有节点上安装也是很合理的，因此我们把它定义为全局单元。它的服务文件名为 skydns.service，内容如下（在你的虚拟机中它应该已经存在）：

```
[Unit]
Description=SkyDNS

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill dns
ExecStartPre=-/usr/bin/docker rm dns
ExecStartPre=/usr/bin/docker pull skynetservices/skydns:2.5.2b
ExecStart=/usr/bin/env bash -c "IP=$(/usr/bin/ip -o -4 addr list docker0 \
  | awk '{print $4}' | cut -d/ -f1) \
  && docker run --name dns -e ETCD_MACHINES=http://$IP:2379 \
     skynetservices/skydns:2.5.2b"
ExecStop=/usr/bin/docker stop dns

[X-Fleet]
Global=true
```

除了 [X-Fleet]，其他部分都只是标准的 systemd 单元文件内容。在 ExecStart 中，我们首先用了一些 Shell 技巧以获取 docker0 网桥的 IP 地址，用于访问主机的 etcd 实例。容器启动时没有用上 -d 参数，这样做可以让 systemd 监控应用程序，以及负责日志记录的工作。[X-Fleet] 部分告诉 fleet，我们希望在所有机器上运行这个单元，而不是按照默认方式只运行在一个实例上。

在启动 DNS 服务器之前，还需要在 etcd 中添加一些配置：

```
core@core-01 ~ $ etcdctl set /skydns/config \
  '{"dns_addr":"0.0.0.0:53", "domain":"identidock.local."}'
{"dns_addr":"0.0.0.0:53", "domain":"identidock.local."}
```

这个配置指示 SkyDNS 负责 identidock.local 域名。

现在可以启动服务了。通过 fleetctl start 命令启动单元文件：

```
core@core-01 ~ $ fleetctl start skydns.service
Triggered global unit skydns.service start
```

list-units 命令可以获取所有单元的状态 。当所有服务都运行起来后，便会得到像这样的输出结果：

```
core@core-01 ~ $ fleetctl list-units
UNIT    MACHINE       ACTIVE  SUB
skydns.service  16aacf8b.../172.17.8.103  active  running
skydns.service  39b02496.../172.17.8.102  active  running
skydns.service  eb570763.../172.17.8.101  active  running
```

从输出结果可以看到，集群中的每台机器上都运行着一个 SkyDNS 容器。

DNS 已经运行起来了，现在需要启动 Redis 容器，并把它注册到 DNS。Redis 的配置就在 redis.service 单元文件中，内容如下：

```
[Unit]
Description=Redis
After=docker.service
Requires=docker.service
After=flanneld.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill redis
ExecStartPre=-/usr/bin/docker rm redis
ExecStartPre=/usr/bin/docker pull redis:3
ExecStart=/usr/bin/docker run --name redis redis:3
ExecStartPost=/usr/bin/env bash -c 'sleep 2 \
  && IP=$(docker inspect -f {{.NetworkSettings.IPAddress}} redis) \
  && etcdctl set /skydns/local/identidock/redis \
     "{\\"host\\":\\"$IP\\",\\"port\\":6379}"'
ExecStop=/usr/bin/docker stop redis
```

这一次的配置里没有包含 `[X-Fleet]` 部分，因此只会启动一个 Redis 实例。在 `[ExecStartPost]` 中，我们提供了一些代码，作用是当容器启动后，Redis 会被自动注册到 SkyDNS。命令中需要用到一个短暂的 `sleep`，目的是在取得 IP 地址之前，先让 Docker 把网络设置好。这种代码通常最好放置在一个辅助性质的脚本里，但为了简单起见，我直接把它放在单元文件中。

现在启动 Redis 服务和 dnmonster 服务（dnmonster 的单元文件与 Redis 的单元文件形式相同）：

```
core@core-01 ~ $ fleetctl start redis.service
Unit redis.service launched on 53a8f347.../172.17.8.101
core@core-01 ~ $ fleetctl start dnmonster.service
Unit dnmonster.service launched on ce7127e7.../172.17.8.102
```

可以看到 dnmonster 和 Redis 的单元被安排在不同的机器上运行，这是为了把负载分摊：

```
core@core-01 ~ $ fleetctl list-units
UNIT      MACHINE       ACTIVE    SUB
dnmonster.service 39b02496.../172.17.8.102  activating  start-pre
redis.service   16aacf8b.../172.17.8.103  activating  start-pre
skydns.service    16aacf8b.../172.17.8.103  active    running
skydns.service    39b02496.../172.17.8.102  active    running
skydns.service    eb570763.../172.17.8.101  active    running
```

机器需要一些时间来下载和启动相关容器。

现在来启动 identidock 容器。它的单元文件 identidock.service 内容如下：

```
[Unit]
Description=identidock

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill identidock
ExecStartPre=-/usr/bin/docker rm identidock
ExecStartPre=/usr/bin/docker pull amouat/identidock:1.0
```

```
ExecStart=/usr/bin/env bash -c "docker run --name identidock --link dns \
  --dns $(docker inspect -f {{.NetworkSettings.IPAddress}} dns) \
  --dns-search identidock.local amouat/identidock:1.0"
ExecStop=/usr/bin/docker stop identidock
```

这一次使用了 Docker 的 `--dns` 和 `--dns-search` 参数，告诉容器通过机器上的 SkyDNS 容器来解析 DNS 查询。为了简单起见，还可以要求 fleet 安排容器运行在当前登录的机器上。首先需要执行 `fleetctl list-machines -l` 来找出机器的 ID：

```
core@core-01 ~ $ fleetctl list-machines -l
MACHINE         IP    METADATA
16aacf8ba9524e368b5991a04bf90aef  172.17.8.103 -
39b02496db124c3cb11ba88a13684c16  172.17.8.102 -
eb570763ac8349ec927fac657bffa9ee  172.17.8.101 -
```

然后在 identidock.service 文件的底部添加以下内容：

```
[X-Fleet]
MachineID=<id>
```

把其中的 `<id>` 更换为你正在运行的机器的 ID。我这里的例子需要改成这样：

```
[X-Fleet]
MachineID=eb570763ac8349ec927fac657bffa9ee
```

现在可以启动 identidock 的单元了，它应该会被安排在当前的机器上运行：

```
core@core-01 ~ $ fleetctl start identidock.service
Unit identidock.service launched on eb570763.../172.17.8.101
```

当服务运行起来后，可以试试是否一切正常工作：

```
core@core-01 ~ $ docker exec -it identidock bash
uwsgi@ae8e3d7c494a:/app$ ping redis
PING redis.identidock.local (192.168.76.3): 56 data bytes
64 bytes from 192.168.76.3: icmp_seq=0 ttl=60 time=1.641 ms
64 bytes from 192.168.76.3: icmp_seq=1 ttl=60 time=2.133 ms
^C--- redis.identidock.local ping statistics ---
2 packets transmitted, 2 packets received, 0% packet loss
round-trip min/avg/max/stddev = 1.641/1.887/2.133/0.246 ms
uwsgi@ae8e3d7c494a:/app$ curl localhost:9090
<html><head><title>Hello
```

还可以测试当一台机器发生故障时会有什么情况：

```
core@core-01 ~ $ fleetctl list-units
UNIT      MACHINE       ACTIVE  SUB
dnmonster.service 39b02496.../172.17.8.102  active  running
identidock.service  eb570763.../172.17.8.101  active  running
redis.service   16aacf8b.../172.17.8.103  active  running
skydns.service    16aacf8b.../172.17.8.103  active  running
skydns.service    39b02496.../172.17.8.102  active  running
skydns.service    eb570763.../172.17.8.101  active  running
```

Redis 运行在 `172.17.8.103`，即名字为 `core-03` 的机器上，可以用 Vagrant 来停止它：

```
core@core-01 ~ $ exit
$ vagrant halt core-03
==> core-03: Attempting graceful shutdown of VM...
```

现在重新登录，然后检查服务状态：

```
core@core-01 ~ $ fleetctl list-units
UNIT      MACHINE       ACTIVE    SUB
dnmonster.service 39b02496.../172.17.8.102  active    running
identidock.service  eb570763.../172.17.8.101  active    running
redis.service   39b02496.../172.17.8.102  activating  start-pre
skydns.service    39b02496.../172.17.8.102  active    running
skydns.service    eb570763.../172.17.8.101  active    running
```

Redis 服务已被自动重新安排在一台运作中的机器上运行。主机下载容器需要一定时间，不过一旦完成后，它就会把新地址注册到 SkyDNS，identidock 也会继续工作。如果在每台主机上预先下载所需的镜像，那就能很大程度上避免等待。

可以看到，fleet 有很多有用的功能，但它比较倾向于长期运行的服务，而不是那种类似批量任务的短期容器。调度策略也非常基本——虽然“最小负载”策略在很多情况下适用，但在某些场景下却要求更加精细或更加复杂的策略。在这种情况下，你可能会发现 Kubernetes 更合适，而且它还可以运行在 fleet 之上。

12.1.3 Kubernetes

Kubernetes（http://kubernetes.io/）是谷歌基于过去十几年实际应用容器的经验而开发出来的容器编排工具。Kubernetes 比较坚持自己的一套设计理念，它对容器的组成和联网方式有强制要求。接下来是一些必须清楚的基本概念。

pod

pod 是一组容器，会被一同安排部署和调度。pod 在 Kubernetes 中被用作调度时不可分割的一个单元，这种做法与其他只有单一容器概念的系统形成了鲜明对比。一个 pod 通常包含 1~5 个互相合作的容器，用于提供某种服务。除了这些用户容器外，Kubernetes 还会运行其他用于提供日志记录和监控服务的容器。pod 在 Kubernetes 中的生命周期是短暂的；随着系统发展，它们会被不断创建和销毁，对此你要有心理准备。

扁平的网络空间

Kubernetes 的联网方式与默认的 Docker 网桥联网相比有着明显的不同。在默认的 Docker 联网中，容器的网络是私有的子网，不能与其他主机上的容器直接通信，除非通过端口转发或代理。在 Kubernetes 中，一个 pod 内的容器共同使用一个 IP 地址，但对于所有 pod，它们的地址空间是“扁平”的，也就是说，所有 pod 无需任何网络地址转换（network address translation，NAT）就能互相通信。这使得多主机集群更易于管理，而代价是 Docker 连接无法使用，并且单台主机（或更准确地说，单个 pod）的联网也变得复杂。由于同一个 pod 内的容器使用相同的 IP，它们可以使用 `localhost` 地址上的端口进行通信（这意味着你需要协调 pod 内的端口使用）。

标签（label）

标签是一组附于 Kubernetes 的对象的键值对，对象主要是 pod，用于描述识别对象的特征（例如 `version: dev` 和 `tier: frontend`）。标签通常不是唯一的，因为标签的设计是用于识别容器群组的。*标签选择符*（label selector）可以用来识别一些对象或对象群（例如，所有环境被设置为生产环境的前端 pod）。通过标签的使用，我们很容易就能够实现分组，譬如把 pod 分配到负载均衡的群组，或者把 pod 在不同群组之间迁移。

服务

服务是一些稳定的、可通过名称定址的端点。服务可以通过标签选择符连接至 pod。例如，我的“缓存”服务可以连接至多个以标签选择符 `"type": "redis"` 表示的“redis” pod。这个服务将以自动循环的方式安排其中一个 pod 处理请求。以这种方式，服务可以把系统的某些部分互相连接。服务提供了一个抽象层，使应用程序只需调用它们而无需了解它们的内部运作。例如，一个在 pod 内运行的应用程序，调用数据库服务时只需知道它的名称和端口，而无需担心数据库由多少个 pod 组成，或上一次使用的是哪一个 pod。Kubernetes 还会为集群设置一个 DNS 服务器，用于监视新的服务，以及允许它们在应用程序代码和配置文件中以名称定址。

另外，服务不一定由 pod 提供，它可以连接到其他已存在的服务，例如外部 API 或数据库。

复制控制器（replication controller）

复制控制器是 Kubernetes 中对 pod 进行实例化的一般做法（通常在 Kubernetes 中不会用到 Docker 的命令行）。它负责控制和监视服务中运行的 pod（称为副本，replica）的数量。例如，假设复制控制器为 Redis 服务时需要保持 5 个 pod 运行，如果其中一个发生故障，那么便会立即启动一个新的副本。如果副本的数量需要减少，它会停止多余的 pod。虽然通过复制控制器进行 pod 的实例化将额外增加一重配置，但同时也大大提升了容错性和可靠性。

图 12-3 中显示的是 Kubernetes 集群的一部分，其中有两个由复制控制器创建的 pod，以及一个为它们提供对外接口的服务。服务将按照 `tier` 标签选择合适的一组 pod，并以循环方式把请求转发至它们中的一个。一个 pod 中的所有容器使用同一个 IP 地址，如果它们需要互相通信，可以通过 `localhost` 地址上的不同端口进行。服务已分配了一个独立的 IP 地址并允许公开访问。

为了使 identidock 能够在 Kubernetes 上运行，我们将会为 dnmonster、identidock 和 Redis 容器分配不同的 pod。这听起来似乎过了头，但这些服务都有单独扩展的可能性（譬如一个 identidock 服务需要两个 dnmonster 服务和三个 Redis 服务，而且它们都能做到负载均衡）。我们不必为了通过 `localhost` 地址上的端口访问服务而重写程序，也无需添加日志记录或监控容器，因为 Kubernetes 已经帮我们做了这些事。Kubernetes 还提供了一个负载均衡的前端代理，因此也不需要 Nginx 的代理容器。

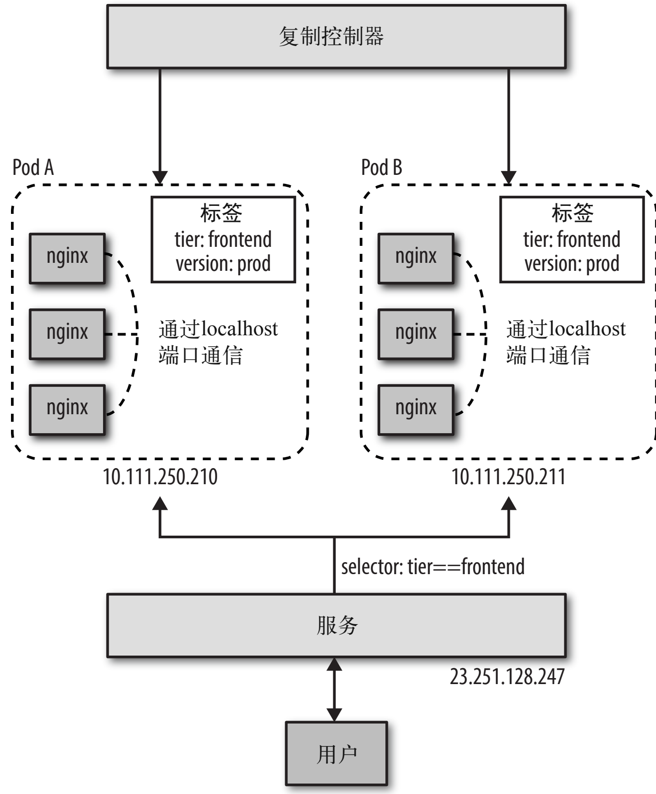


图 12-3：Kubernetes 集群示例

获取 Kubernetes

Kubernetes 的 GitHub 页面上提供了不同平台的入门指南。如果你希望在本地体验 Kubernetes，Kubernetes 的 GitHub 页面上提供了一些 Docker 容器（https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/getting-started-guides/docker.md）及 Vagrant 虚拟机（https://github.com/GoogleCloudPlatform/kubernetes/blob/master/docs/getting-started-guides/vagrant.md），你可以用它们来运行 Kubernetes。除此以外，一个比较理想的 Kubernetes 托管方案是谷歌的容器引擎（Google Container Engine，GKE，https://cloud.google.com/container-engine/），它是谷歌自己的一套商业产品。

如果你选择自行安装 Kubernetes，你还需要配置 DNS 插件，否则无法解析服务名称；而 GKE 已经配置好这些并随时可用。

下面的指令假设 Kubernetes 是在谷歌容器引擎（GKE）上运行，但对于其他的 Kubernetes 安装来说应该区别不大。[2] 有关获取或安装 Kubernetes 的详情参见上面的辅助栏“获取 Kubernetes”。本节的余下部分假设你已经可以成功运行 kubectl 命令，而且 DNS 服务器也已经一并安装妥当。

下面来定义一个用于创建 Redis 实例的复制控制器。首先创建一个名为 redis-controller.json 的文件，内容如下：

```
{ "kind":"ReplicationController",
   "apiVersion":"v1",
   "metadata":{ "name":"redis-controller" },
   "spec":{
      "replicas":1,
      "selector":{ "name":"redis-pod" },
      "template":{
         "metadata":{
            "labels":{ "name":"redis-pod" }
         },
         "spec":{
            "containers":[ {
                  "name":"redis",
                  "image":"redis:3",
                  "ports":[ {
                        "containerPort":6379,
                        "protocol":"TCP"
                     } ] } ] } } } }
```

这里要求 Kubernetes 创建一个以复制控制器管理的 pod，它包含一个运行 redis:3 镜像的容器，并开放了 6379 端口。我们把一个键名为“name”，值为“redis-pod”的标签赋予这个 pod。复制控制器本身也是一个对象，名为“redis-controller”。

现在通过 kubectl 工具启动这个 pod：

```
$ kubectl create -f redis-controller.json
services/redis
```

如果接着执行 kubectl get pods，你将获得一个列表，列出运行中和待运行的所有 pod，其中包含各种细节，有标签和 IP 地址，以及它们使用的容器和镜像 [3]。由于命令输出的信息太多，这里不便全部列出，但其中的运行镜像部分大概是这样的：

```
gcr.io/google_containers/fluentd-gcp:1.6
gcr.io/google_containers/skydns:2015-03-11-001
gcr.io/google_containers/kube2sky:1.9
gcr.io/google_containers/etcd:2.0.9
redis:3
```

redis 以外的其他容器负责执行各种系统任务；fluentd 负责日志记录，skydns、kube2sky 和 etcd 负责处理服务名称的 DNS 解析。请注意，我们的 Redis pod 处于待运行状态，它需要一些时间让 Kubernetes 下载镜像和启动 pod。

注 2：这里的范例使用的是 v1 的 API。预计后续版本的 API 语法将略有不同。

注 3：同样，你可以执行 kubectl get rc 获得复制控制器的列表，以及 kubectl get services 获得所有服务的列表。

如果你正在使用 GKE，并想了解它在背后如何运作，可以执行 `gcloud compute ssh HOST` 登录到 pod 的虚拟机中，其中的 `HOST` 是 `kubectl get pods` 命令的输出中 HOST 标题的值（只使用 / 之前的部分）。登录后可以执行 `docker ps`，就像在一般运行 Docker 容器的虚拟机上一样，与容器进行交互。

下一步就是定义一个服务，让其他容器无需知道 IP 地址，就能够连接到我们的 redis pod。把以下内容保存到 redis-service.json 文件中：

```
{ "kind":"Service",
   "apiVersion":"v1",
   "metadata":{ "name":"redis" },
   "spec":{
      "ports": [ {
          "port":6379,
          "targetPort":6379,
          "protocol":"TCP"
        } ],
      "selector":{ "name":"redis-pod" }
   } }
```

这个配置文件定义了一个服务，让使用这个服务的程序能够连接到我们的 redis pod。这项服务被命名为“redis”，DNS 集群插件（默认已安装在 GKE 上）能够发现它，并负责这个名字的 DNS 解析。重要的是，这意味着我们的 identidock 代码无需修改主机名就能继续工作。

这个 redis pod 以 `"name":"redis-pod"` 选择符识别。如果我们有多个相同标签的 redis 节点，选择符将会匹配所有节点。当有一个以上的 pod 被选中时，服务将随机选择一个 pod 处理请求（也可以在选择 pod 的时候设置*关联度*。例如，`"ClientIP"` 将始终根据 IP 地址把 pod 分配给客户端）。通过修改 pod 的标签，它们可以动态地把 pod 移入或移出相关的选择符群组，有些任务需要这样做，比如为了对 pod 进行调试或维护，需要暂时把 pod 从生产环境中撤离。

接下来用几乎相同的方式创建 dnmonster 控制器和服务。把以下内容保存到 dnmonster-controller.json 文件中：

```
{ "kind":"ReplicationController",
   "apiVersion":"v1",
   "metadata":{ "name":"dnmonster-controller" },
   "spec":{
      "replicas":1,
      "selector":{ "name":"dnmonster-pod" },
      "template":{
         "metadata":{
            "labels":{ "name":"dnmonster-pod" } },
         "spec":{
            "containers":[ {
                  "name":"dnmonster",
                  "image":"amouat/dnmonster:1.0",
                  "ports":[ {
                        "containerPort":8080,
                        "protocol":"TCP"
                     } ] } ] } } } }
```

为 dnmonster 服务创建 dnmonster-service.json 文件：

```
{ "kind":"Service",
   "apiVersion":"v1",
   "metadata":{ "name":"dnmonster" },
   "spec":{
      "ports": [ {
          "port":8080,
          "targetPort":8080,
          "protocol":"TCP"
        } ],
      "selector":{ "name":"dnmonster-pod" }
   } }
```

启动它们：

```
$ kubectl create -f dnmonster-controller.json
replicationcontrollers/dnmonster-controller
$ kubectl create -f dnmonster-service.json
services/dnmonster
```

这部分与处理 redis 控制器和服务的手法完全相同，我们有一个可以通过主机名 `dnmonster` 访问的 dnmonster 服务，它把请求转发至由复制控制器创建的 `dnmonster` 实例。

现在可以创建一个 identidock 的 pod，把刚才创建的所有东西连在一起。把以下内容保存到 identidock-controller.json 文件：

```
{ "kind":"ReplicationController",
   "apiVersion":"v1",
   "metadata":{ "name":"identidock-controller" },
   "spec":{
      "replicas":1,
      "selector":{ "name":"identidock-pod" },
      "template":{
         "metadata":{
            "labels":{ "name":"identidock-pod" } },
         "spec":{
            "containers":[ {
                  "name":"identidock",
                  "image":"amouat/identidock:1.0",
                  "ports":[ {
                        "containerPort":9090,
                        "protocol":"TCP"
                     } ] } ] } } } }
```

并启动它：

```
$ kubectl create -f identidock-controller.json
replicationcontrollers/identidock-controller
```

Identidock 现在应该已经运行起来了，但是仍然需要创建一个 identidock 服务，让它可以被外界访问。把以下内容保存到 identidock-service.json 文件：

```
{ "kind":"Service",
   "apiVersion":"v1",
```

```
    "metadata":{ "name":"identidock" },
    "spec":{
       "type": "LoadBalancer",
       "ports": [ {
           "port":80,
           "targetPort":9090,
           "protocol":"TCP"
         } ],
       "selector":{ "name":"identidock-pod" }
    } }
```

这个服务与之前定义的有些许不同。我们把 "type" 设置为 "LoadBalancer"，这将创建一个外界能访问的负载均衡器，它会在 80 端口上监听连接，并将连接转发至 9090 端口的 identidock 服务。

如果你使用 GKE，那么可能还需要在防火墙打开 80 端口，通过 gcloud 工具创建规则便可：

```
$ gcloud compute firewall-rules create --allow=tcp:80 identidock-80
```

现在，假设我们与服务之间没有防火墙，你应该能够通过 identidock 服务显示的公共 IP 地址连接它：

```
$ kubectl get services identidock
NAME          LABELS     SELECTOR               IP(S)            PORT(S)
identidock    <none>     name=identidock-pod   10.111.250.210   80/TCP
                                               23.251.128.247
$ curl 23.251.128.247
<html><head><title>Hello...
```

Kubernetes 数据卷

数据卷的使用在 Kubernetes 中也有所不同。主要的区别在于，它们在 pod 这一层声明，而不是在容器层，并且可以与 pod 内的容器共享。Kubernetes 为不同类型的用例提供了多种数据卷，列举如下。

emptyDir

它会在容器中初始化一个允许容器写入的空目录。当 pod 不再存在时，目录也会一并消失。这种类型很适合 pod 存在时才需要的临时数据，或者会定期备份到持久性存储的数据。

gcePersistentDisk

对于 GKE 用户而言，这种类型可以把数据存储在谷歌云内。数据的保存将超越 pod 的生命周期。

awsElasticBlockStore

对于 AWS 用户而言，这种类型可以把数据存储在 Amazon 的弹性块存储（Elastic Block Store，EBS）内。数据的保存将超越 pod 的生命周期。

nfs

适用于网络文件系统（Network File System，NFS）上的文件。同样，数据的保存将超越 pod 的生命周期。

secret

适用于存储敏感信息，如密码和 pod 使用的 API 令牌。这种用于保密的数据卷必须通过 Kubernetes API 初始化，并存储在 tmpfs 上，tmpfs 的特性是完全存在于内存中，绝对不会写入磁盘。

使用 Kubernetes 需要更多的配置工作，但如果利用它来搭建系统，将会有现成的故障转移和负载均衡的支持。与容器互联的做法相比，Kubernetes 服务提供了一个抽象层，使我们能够对底层的 pod 和容器轻松地进行扩展和替换。它的缺点在于，Kubernetes 在我们简单的 identidock 应用上增加了相当多的负担，额外的日志记录和监控功能需要大量资源，从而增加了运行成本。

对于某些应用来说，Kubernetes 强制的系统设计和选择并不合适。但对于大多数应用来说，特别是微服务以及状态信息很少或相对独立的应用，只需极少付出，就能得到一个由 Kubernetes 提供的易用、复原能力强，并且可扩展的服务。

12.1.4　Mesos 和 Marathon

Apache Mesos（https://mesos.apache.org/）是一个开源的集群管理工具。它的设计目标是让集群能够扩展到几百至几千台主机的大型集群。Mesos 支持来自不同用户的工作负载，而这些负载更可以是千差万别的，例如某用户的 Docker 容器可以在另一位用户的 Hadoop 任务旁一同运行。

Apache Mesos 始于加州大学伯克利分校的一个项目，后来成为驱动 Twitter 的底层基础架构，在许多大公司如 eBay 和 Airbnb 里也是一个重要的工具。很多 Mesos 的持续开发工作和支撑工具（如 Marathon）都是由 Mesos 的其中一位原开发者 Ben Hindman 所创立的 Mesosphere 公司承担。

Mesos 的架构围绕高可用性和高度复原能力而设计。Mesos 集群的主要组成部分列举如下。

Mesos 代理节点（agent node）[4]

负责实际运行任务。所有代理把它们的可用资源提交给 master。一般会有几十到上千个代理节点。

Mesos master

负责把任务分发至代理，并维护一个可用资源列表，以供框架使用。master 按分配策略决定可提供多少资源。通常有两个或四个备用的 master，以备故障发生时可以随时替换。

注 4：过去称为 slave 节点。

ZooKeeper

用于选举以及查找当前 master 的地址。通常有三个或五个 ZooKeeper 实例处于运行状态，以确保高可用性以及应对故障发生。

框架（framework）

框架通过与 master 协调，把任务安排到代理节点上运行。框架由两部分组成：executor 和 scheduler，前者运行在代理之中，负责执行任务，后者需要通过 master 注册，并根据 master 提供的可用资源信息来选择使用的资源。在一个 Mesos 集群中，为了处理不同类型的任务，可能有多个框架运行着。任务的提交应该向框架提出，而不应直接向 Mesos 提交。

图 12-4 中展示了一个使用 Marathon 框架作为调度器的 Mesos 集群。Marathon 调度器通过 ZooKeeper 找到当前的 Mesos master，并将任务提交给它。Marathon 调度器和 Mesos master 都有后备的进程运行着，以备 master 不可用时还能准备开始工作。

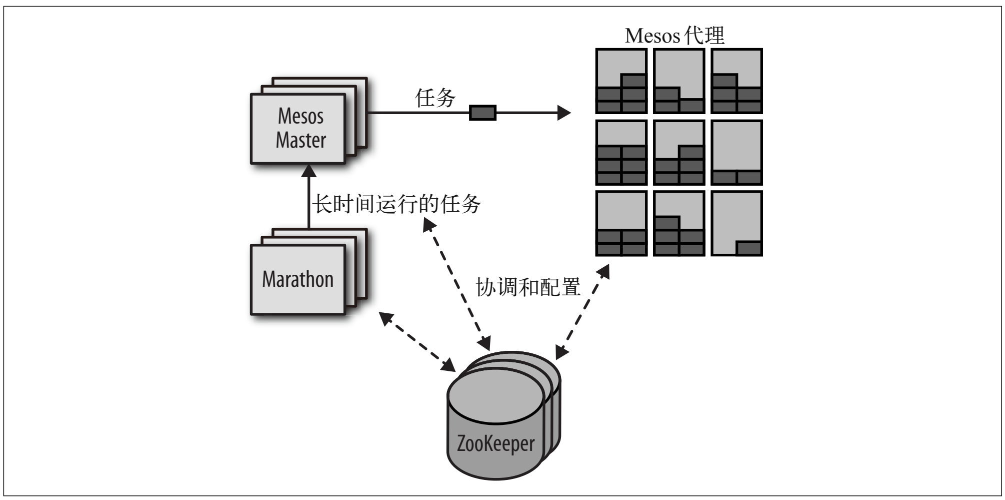


图 12-4：Mesos 集群

通常 ZooKeeper 与 Mesos master 和它的后备进程运行在同一台主机上。在小型的集群中，主机上还可以运行代理，但规模较大的集群则需要与 master 进行通信，使得这样做不太可行。Marathon 也可以在相同的主机上运行，或运行在网络边界的其他主机上，成为客户端的接入点，从而使客户端与 Mesos 集群本身分开。

Marathon（https://mesosphere.github.io/marathon/，出自 Mesosphere）针对长时间运行[5]的应用程序而设计，用于应用的启动、监控和扩展。Marathon 启动应用的设计非常灵活，甚至可以用来启动其他诸如 Chronos（即“cron”的数据中心版本）的辅助框架。Marathon 直接支持 Docker 容器，使它成为一个运行 Docker 容器不错的框架。与之前介绍过的其他编排框架一样，Marathon 支持各种关联度和约束的规则。客户端通过 REST API 与 Marathon

注 5：大概这就是它被称为 Marathon（马拉松）的原因吧，原来如此……

进行交互。其他功能还包括健康检查，以及可用于集成负载均衡器或分析数据的事件流。

为了解 Mesos 和 Marathon 如何工作，可以通过使用 Docker Machine 模仿图 12-4 中的设置，建立一个具有 3 个节点的集群。然而，ZooKeeper、Marathon 调度器和 Mesos master 都只会各运行一个实例，而所有节点将运行 Mesos 代理。用于生产环境的架构与此区别较大，核心服务都应运行多个实例以确保其高可用性。

首先来创建主机，分别是 mesos-1、mesos-2 和 mesos-3：

```
$ docker-machine create -d virtualbox mesos-1
Creating VirtualBox VM...
...
$ docker-machine create -d virtualbox mesos-2
...
$ docker-machine create -d virtualbox mesos-3
...
```

这里需要一点修改，使 mesos 的主机名能够被解析。理论上这是不需要的，但不这样做的话，运行时会遇到一些问题：

```
$ docker-machine ssh mesos-1 'sudo sed -i "\$a127.0.0.1 mesos-1" /etc/hosts'
$ docker-machine ssh mesos-2 'sudo sed -i "\$a127.0.0.1 mesos-2" /etc/hosts'
$ docker-machine ssh mesos-3 'sudo sed -i "\$a127.0.0.1 mesos-3" /etc/hosts'
```

我们将首先对 mesos-1 进行配置，把 Mesos master、ZooKeeper、Marathon 框架以及代理运行起来。

第一个需要启动的是 ZooKeeper，因为其他容器需要它来进行注册和查找服务。这里所用的是一个我自己创建的镜像，因为直到我写这一部分的时候，还没有可用的官方镜像。启动这个容器的时候，我用了 --net=host 参数，主要是出于效率的考虑，以及与 master 和代理容器保持一致，因为它们需要主机联网功能才能打开新的端口来提供服务。

```
$ eval $(docker-machine env mesos-1)
$ docker run --name zook -d --net=host amouat/zookeeper
...
Status: Downloaded newer image for amouat/zookeeper:latest
dfc27992467c9563db05af63ecb6f0ec371c03728f9316d870bd4b991db7b642
```

为了使之后的配置工作更容易一些，我们把节点的 IP 地址保存到变量中：

```
$ MESOS1=$(docker-machine ip mesos-1)
$ MESOS2=$(docker-machine ip mesos-2)
$ MESOS3=$(docker-machine ip mesos-3)
```

现在可以启动 master：

```
$ docker run --name master -d --net=host \
             -e MESOS_ZK=zk://$MESOS1:2181/mesos \  ❶
             -e MESOS_IP=$MESOS1 \  ❷
             -e MESOS_HOSTNAME=$MESOS1 \
             -e MESOS_QUORUM=1 \  ❸
             mesosphere/mesos-master:0.23.0-1.0.ubuntu1404  ❹
...
```

```
Status: Downloaded newer image for mesosphere/mesos-master:0.23.0-1.0.ubuntu1404
9de83f40c3e1c5908381563fb28a14c2e23bb6faed569b4d388ddfb46f7d7403
```

❶ 把 ZooKeeper 的位置告诉 master，并进行注册。
❷ 设置 master 的 IP 地址。
❸ 由于我们只是为了演示，这里只有一个 master 节点。
❹ 使用来自 Mesosphere 的 mesos 镜像。因为这个镜像并非自动构建的，很难说它内部到底包含了什么东西，所以不建议在生产环境中使用它。

同时在同一台主机上执行代理：

```
$ docker run --name agent -d --net=host \
             -e MESOS_MASTER=zk://$MESOS1:2181/mesos \
             -e MESOS_CONTAINERIZERS=docker \  ❶
             -e MESOS_IP=$MESOS1 \
             -e MESOS_HOSTNAME=$MESOS1 \
             -e MESOS_EXECUTOR_REGISTRATION_TIMEOUT=10mins \  ❷
             -e MESOS_RESOURCES="ports(*):[80-32000]" \  ❸
             -e MESOS_HOSTNAME=$MESOS1 \
             -v /var/run/docker.sock:/run/docker.sock \  ❹
             -v /usr/local/bin/docker:/usr/bin/docker \
             -v /sys:/sys:ro \  ❺
             mesosphere/mesos-slave:0.23.0-1.0.ubuntu1404
...
Status: Downloaded newer image for mesosphere/mesos-slave:0.23.0-1.0.ubuntu1404
38aaec1d08a41e5a6deeb62b7b097254b5aa2b758547e03c37cf2dfc686353bd
```

❶ Mesos 有一个概念称为 containerizers，它为不同任务之间提供隔离，且运行于代理上。这里以 docker 作为参数，表示 Docker 容器将以任务形式运行在代理上。
❷ 我们需要延长"注册超时时间"，让代理有足够时间下载镜像。
❸ 默认情况下，代理只把部分高端口提供给框架使用。由于 identidock 需要使用一些低端口，我们必须明确地将它们在可提供的资源中列出。为了简明起见，我并没有刻意排除其中 Mesos 用到的端口，但如果框架要求使用一个已被占用的端口的话，这样做将可能导致冲突。
❹ 为了使代理能够启动新的容器，需要加载 Docker 的 sock 和二进制文件。
❺ 加载 /sys 是必要的，否则代理无法准确和详细地报告主机的可用资源。

```
$ docker run -d --name marathon -p 9000:8080 \  ❶
             mesosphere/marathon:v0.9.1 --master zk://$MESOS1:2181/mesos \
                 --zk zk://$MESOS1:2181/marathon \
                 --task_launch_timeout 600000  ❷
...
Status: Downloaded newer image for mesosphere/marathon:v0.9.1
697d78749c2cfd6daf6757958f8460963627c422710f366fc86d6fcdce0da311
```

❶ 我们需要更改 Marathon 的默认 8080 端口，以避免可能与 dnmonster 容器发生冲突。
❷ 把超时设置成 600 000 毫秒，以配合代理的 executor 注册时的 10 分钟超时时间。这个设置只是暂时的，未来的版本将会取消这一设置。

下一步，在其他主机上启动代理：

```
$ eval $(docker-machine env mesos-2)
$ docker run --name agent -d --net=host \
            -e MESOS_MASTER=zk://$MESOS1:2181/mesos \
            -e MESOS_CONTAINERIZERS=docker \
            -e MESOS_IP=$MESOS2 \
            -e MESOS_HOSTNAME=$MESOS2 \
            -e MESOS_EXECUTOR_REGISTRATION_TIMEOUT=10mins \
            -e MESOS_RESOURCES="ports(*):[80-32000]" \
            -v /var/run/docker.sock:/run/docker.sock \
            -v /usr/local/bin/docker:/usr/bin/docker \
            -v /sys:/sys:ro \
            mesosphere/mesos-slave:0.23.0-1.0.ubuntu1404
...
Status: Downloaded newer image for mesosphere/mesos-slave:0.23.0-1.0.ubuntu1404
ac1216e7eedbb39475404f45a5655c7dc166d118db99072ed3d460322ad1a1c2
$ eval $(docker-machine env mesos-3)
$ docker run --name agent -d --net=host \
            -e MESOS_MASTER=zk://$MESOS1:2181/mesos \
            -e MESOS_CONTAINERIZERS=docker \
            -e MESOS_IP=$MESOS3 \
            -e MESOS_HOSTNAME=$MESOS3 \
            -e MESOS_EXECUTOR_REGISTRATION_TIMEOUT=10mins \
            -e MESOS_RESOURCES="ports(*):[80-32000]" \
            -v /var/run/docker.sock:/run/docker.sock \
            -v /usr/local/bin/docker:/usr/bin/docker \
            -v /sys:/sys:ro \
            mesosphere/mesos-slave:0.23.0-1.0.ubuntu1404
...
Status: Downloaded newer image for mesosphere/mesos-slave:0.23.0-1.0.ubuntu1404
b5eecb7f56903969d1b7947144617050f193f20bb2a59f2b8e4ec30ef4ec3059
```

现在，如果在浏览器中打开 http://$MESOS1:5050（把 $MESOS1 替换为 `mesos-1` 的 IP 地址），应该可以看到 Mesos 的 Web 界面。同样，Marathon 的界面可以通过 9000 端口访问。

我们现在已经有一个通过 Marathon 在 Mesos 代理上运行容器的基础设施。但我们还需要添加一个服务发现的机制，才能够让 identidock 运行起来。为此，我们将在 `mesos-1` 上启用 `mesos-dns`（https://github.com/mesosphere/mesos-dns）。

Marathon 的任务以 JSON 文件定义，对于这个将要启动的任务，JSON 文件包含它的详细信息以及所需资源。

可以利用以下的 JSON 文件启动 `mesos-1` 的 `mesos-dns`：

```
{
  "id": "mesos-dns",
  "container": {
    "docker": {
      "image": "bergerx/mesos-dns",  ❶
      "network": "HOST",  ❷
      "parameters": [
          { "key": "env",
            "value": "MESOS_DNS_ZK=zk://192.168.99.100:2181/mesos" },  ❸
          { "key": "env", "value": "MESOS_DNS_MASTERS=192.168.99.100:5050" },
          { "key": "env", "value": "MESOS_DNS_RESOLVERS=8.8.8.8" }
        ]
    }
```

```
  },
  "cpus": 0.1, ❹
  "mem": 120.0,
  "instances": 1, ❺
  "constraints": [["hostname", "CLUSTER", "192.168.99.100"]] ❻
}
```

❶ 这里使用一个由普通用户构建的 `mesos-dns`，因为它能够自动从环境变量读取配置，本书写作之际，`mesosphere/mesos-dns` 镜像还不支持这个功能。

❷ 与之前一样，为了效率而使用主机联网模式是合理的做法，虽然也可以使用网桥联网模式并同时发布 53 端口。

❸ 我们需要设置环境变量来配置 `mesos-dns`。可以通过使用 `parameter` 选项做到这一点，`parameter` 用于给启动镜像的 `docker run` 命令添加参数。务必把 `192.168.99.100` 更换为集群中 `mesos-1` 的 IP 地址。

❹ 所有任务都需要定义运行时所需的资源。这里要求“0.1”的 CPU 资源和 120MB 的内存。

❺ 在这个测试中，只需要一个 `mesos-dns` 的实例。

❻ 通过以 `mesos-1` 的 IP 地址指定一个主机名（hostname）的约束条件，我们能够把 `mesos-dns` 固定在 `mesos-1` 主机上。

实际上，资源如何分配给容器，取决于 Mesos 使用的 isolator，isolator 是个可配置项。通常情况下，CPU 资源所使用的是一个相对的权重值，即当出现争用 CPU 的情况时，权重值为 0.2 的容器能够使用的 CPU 将两倍于权重值为 0.1 的容器。代理也会自行从提供给 Mesos 的资源中扣除 CPU 的数值（因此，如果某代理有“8”个 CPU 的资源，且运行着“1”个 CPU 的任务时，对往后的任务它只会提供“7”个 CPU 的资源）。

将上述文件保存为 dns.json，并通过以下命令调用 REST API 来把文件发送给 Marathon：

```
$ curl -X POST http://$MESOS1:9000/v2/apps -d @dns.json \
      -H "Content-type: application/json" | jq .
{
  "id": "/mesos-dns",
  "cmd": null,
  "args": null,
  "user": null,
...
```

还可以使用 `marathonctl` 命令行工具或 Web 界面提交任务。

如果现在查看 Marathon 的 Web 界面，应该能够看到它正在部署 `mesos-dns`。当 `mesos-dns` 运行起来后，我们需要告诉每一台主机使用它。最简单的办法便是更新每台主机上的 resolv.conf，当容器启动时便会自动获得。

可以在每台主机上运行这个 `sed` 脚本：

```
$ docker-machine ssh mesos-1 \
  "sudo sed -i \"1s/^/domain marathon.mesos\nnameserver $MESOS1\n/\" \
   /etc/resolv.conf"
$ docker-machine ssh mesos-2 \
  "sudo sed -i \"1s/^/domain marathon.mesos\nnameserver $MESOS1\n/\" \
   /etc/resolv.conf"
```

```
$ docker-machine ssh mesos-3 \
  "sudo sed -i \"1s/^/domain marathon.mesos\nnameserver $MESOS1\n/\" \
   /etc/resolv.conf"
```

运行之后，resolv.conf 将会变成这样：

```
domain marathon.mesos
nameserver 192.168.99.100
...
```

如果你的 resolv.conf 包括一行 `search` 指令，那么你还需要把 marathon.mesos 写进去，否则域名解析将会失败。

现在为每个容器创建 launcher。它们会被放在网桥网络上，但它们必须都各开放一个端口，以便能够通过主机访问它们。

与以往一样，我们会先将 Redis 运行起来。将以下内容保存为 redis.json：

```
{
    "id": "redis",
    "container": {
      "docker": {
        "image": "redis:3",
        "network": "BRIDGE",
        "portMappings": [
          {"containerPort": 6379, "hostPort": 6379}
        ]

      }
    },
    "cpus": 0.3,
    "mem": 300.0,
    "instances": 1
}
```

并提交至 Mesos：

```
$ curl -X POST http://$MESOS1:9000/v2/apps -d @redis.json \
       -H "Content-type: application/json"
...
```

dnmonster 也一样，将以下内容保存为 dnmonster.json：

```
{
    "id": "dnmonster",
    "container": {
      "docker": {
        "image": "amouat/dnmonster:1.0",
        "network": "BRIDGE",
        "portMappings": [
          {"containerPort": 8080, "hostPort": 8080}
        ]

      }
    },
    "cpus": 0.3,
```

```
    "mem": 200.0,
    "instances": 1
}
```

并提交至 Mesos：

```
$ curl -X POST http://$MESOS1:9000/v2/apps -d @dnmonster.json \
       -H "Content-type: application/json"
...
```

最后，将以下内容保存为 identidock.json：

```
{
    "id": "identidock",
    "container": {
      "docker": {
        "image": "amouat/identidock:1.0",
        "network": "BRIDGE",
        "portMappings": [
          {"containerPort": 9090, "hostPort": 80}
        ]

      }
    },
    "cpus": 0.3,
    "mem": 200.0,
    "instances": 1
}
```

并提交至 Mesos：

```
$ curl -X POST http://$MESOS1:9000/v2/apps -d @identidock.json \
       -H "Content-type: application/json"
...
```

一旦代理已成功下载并启动镜像，通过分配 idenidock 任务的主机的 IP 地址，你便能够访问 identidock。你可以从 Web 界面或 REST API 找到 IP 地址，像这样：

```
$ curl -s http://$MESOS1:9000/v2/apps/identidock | jq '.app.tasks[0].host'
"192.168.99.101"
$ curl 192.168.99.101
<html><head><title>Hello...
```

假如有足够的可用资源，Marathon 将确保任何不正常停止的应用程序能够重新启动。你可以尝试停止并重新启动 `mesos-2` 和 `mesos-3` 机器，通过这样做，你可以学习到，当资源离线而新的资源重新上线后，任务是如何迁移和重新启动的。

我们还可以把更复杂的健康检查添加到应用程序中，而且做法很简单，一般是实现一个 Marathon 能够定期轮询的 HTTP 端点。例如，可以把 identidock.json 更新为以下内容：

```
{
    "id": "identidock",
    "container": {
      "docker": {
        "image": "amouat/identidock:1.0",
```

```
            "network": "BRIDGE",
            "portMappings": [
              {"containerPort": 9090, "hostPort": 80}
            ]

          }
        },
        "cpus": 0.3,
        "mem": 200.0,
        "instances": 1,
        "healthChecks": [
            {
                "protocol": "HTTP",
                "path": "/",
                "gracePeriodSeconds": 3,
                "intervalSeconds": 10,
                "timeoutSeconds": 10,
                "maxConsecutiveFailures": 3
            }]
    }
```

以上的做法将尝试每隔 10 秒抓取 identidock 的主页一次。如果由于任何原因抓取失败（例如，返回代码不是介于 200~399，或响应没有在超时时限内收到），Marathon 将进行最多两次测试，之后便会终止任务。

为了部署这个健康检查功能，只需把旧的 identidock 停止，然后把更新后的启动起来：

```
$ curl -X DELETE http://$MESOS1:9000/v2/apps/identidock
{"version":"2015-09-02T13:53:23.281Z","deploymentId":"1db18cce-4b39-49c0-8f2f...
$ curl -X POST http://$MESOS1:9000/v2/apps -d @identidock.json \
       -H "Content-type: application/json"
...
```

如果现在浏览 Marathon 的 Web 界面，应该能够发现“Health Check Results”（健康检查结果）。

通过安排 `mesos-dns` 容器在特定 IP 地址的主机上运行，我们已经看到 Marathon 的约束条件是如何工作的。我们也可以指定一些约束条件来选择具有（或不具有）某些特定属性的主机，例如，为了系统能够支持容错而将容器分散在各个主机上运行。

当前的配置存在一个问题，那就是 identidock 服务的地址，而这取决于它被安排用来运行的主机的 IP。显然，我们非常需要一个能够从静态端点路由至 identidock 服务的可靠方法。方法之一是使用 `mesos-dns` 找出当前端点，除此之外，Marathon 提供了一个名为 servicerouter 的工具（https://github.com/mesosphere/marathon/blob/master/bin/servicerouter.py），它能够生成用于 HAProxy（http://www.haproxy.org/）的配置，使流量路由至 Marathon 的程序。另一个解决方案是创建自己的代理服务器或负载均衡服务，让它们在 Marathon 的事件总线上监听应用程序的创建和销毁事件，并针对不同情况转发请求。

另一个问题就是固定端口的使用，像 Redis 的 6379 端口和 dnsmonster 的 8080 端口。我们已经不得不把 Marathon 的 Web 界面重新映射到 9000 端口，以避免与 dnmonster 发生冲突。为了解决这个问题，我们可以重写程序，使程序使用由 Marathon 分配的动态端口，但

更好的解决办法应该是使用 SDN（软件定义网络）。网上也有一些教程讲述如何将 Weave 和 Mesos 配合使用，Mesospher 也正在开发 Calico 项目（参见 11.5.4 节）与 Mesos 的整合，这也有望成为 Mesos 的原生功能之一。

Marathon 还有一个概念称为*应用程序组*（application group），用于把应用程序组合起来，确保它们按正确的顺序部署，从而在启动、扩展或进行滚动更新时，都能满足依赖关系（例如，数据库需要在应用程序服务器之前启动）。

相比于其他的编排解决方案，Mesos 的一个独特功能是支持混合的工作负载。有了它，在一个集群中可以同时运行多个框架，允许如 Hadoop 或 Storm 等数据处理任务与驱动微服务应用的 Docker 容器一并运行。这是 Mesos 在追求高利用率的环境中特别有用的功能之一；你可以在同一台主机上安排运行一个高 CPU 使用率但低带宽要求的任务，同时又安排运行另一个性质相反的任务，以达到资源使用率的最大化。一个集群的有效使用依赖于是否准确地请求资源，而不是向 Marathon 或其他框架提交任务时过度配置。在 identidock 的范例中，内存使用的数字被刻意增加，目的是为了确保不会出现所有任务都被分配在同一个代理上的情况，虽然在现实中一台主机也足以胜任。为了解决这个问题，Mesos 支持*超额预定*（over-subscription），允许在一些本来没有提供足够资源，但通过监控发现仍有能力运行任务的代理上执行“可撤回的”（revocable）任务。当资源的使用急剧上升时，这些可撤回的任务将被停止，因此它们通常属于低优先级的任务，例如负责在后台进行数据分析。获取更多关于 Mesos 超额预定的信息，请访问 Mesos 网站（http://mesos.apache.org/documentation/latest/oversubscription/）以及 GitHub（https://github.com/mesosphere/serenity）。

Mesos 上运行 Swarm 或 Kubernetes

由于 Mesos 本身提供了一个低阶的集群和调度基础设施，在 Mesos 之上运行更高阶的接口（例如 Kubernetes 和 Swarm）绝对是可行的。乍一看这样做好像很愚蠢，因为它们在功能上有很大程度的重叠。然而，运行在 Mesos 之上，意味着你可以充分利用现有的 Mesos 功能来实现容错、高可用性和有效的资源运用。你还可以借助 Mesos 便于移植的特性，把它移植到任何支持 Mesos 的数据中心或云端，并同时能够保持 Kubernetes 和 Swarm 的功能和易用性。对于运营来说这样做也有明显的益处，它们只需专注于提供底层的 Mesos 基础设施，便能够同时支持多样化的工作负载和高利用率。

想要了解更多信息，请访问 Kubernetes-Mesos 的页面（https://github.com/kubernetes/kubernetes/tree/master/contrib/mesos）。

12.2 容器管理平台

目前有几个可选的容器管理平台，专门针对容器的配置、编排以及监控的自动化而开发。这些平台不直接提供容器托管，而是提供基于公有云和私有云的接口。在即将看到的所有例子中，它们都提供了 Web 界面和系统概览。这些平台对于简化容器管理非常有用，而且在基础设施供应商之上提供了一个抽象层，使云平台之间的迁移或同时使用多个云平台变得更容易。

12.2.1　Rancher

Rancher（http://rancher.com/rancher/），中文的意思是“牧场主”，我猜它的命名应该是来自“宠物与牲口”（pets versus cattle）的比喻，在众多的管理平台之中，它是对 Docker 最具针对性的一个。

Rancher 的入门非常简单，只需要在一台主机上运行 Rancher 服务器的容器：

```
$ docker run -d --restart=always -p 8080:8080 rancher/server
```

然后，在浏览器中打开主机的 8080 端口，就能看到 Rancher 的界面。在网页上，你可以开始添加主机，主机可以是你的架构中的虚拟机或者云端的资源。如果你使用像 Digital Ocean 或 AWS 的公有云，并把访问密钥提供给 Rancher 的话，它便能自动把虚拟机部署妥当。如果选择在现有的虚拟机上安装 Rancher，你只需参照 Rancher 的网页上提供的参数，在主机上执行 Rancher 代理时附上这些参数即可，例子如下：

```
$ docker run -d --privileged -v /var/run/docker.sock:/var/run/docker.sock \
    rancher/agent:v0.8.1 http://<host_ip>:8080/v1/scripts/<token>
```

Rancher 需要加载 Docker 的 sock 文件，之后才能在主机上启动新的容器。当代理启动并运行起来后，它便会出现在 Rancher 页面的 HOSTS 标签页上。基础设施的页面上显示了所有正在主机上运行的容器（除了 Rancher 代理）。如果只是测试的话，Rancher 服务器与代理可以运行在同一台主机上，但 Rancher 建议服务器在生产环境中使用专属的主机。

在 Rancher 上运行 identidock 轻而易举，只需为每个容器创建一个服务，然后把 identidock 服务连接到 dnmonster 和 Redis 服务即可。还可以选择把 identidock 使用的 9000 端口映射到 80 端口。除此以外，没有其他端口需要开放，Rancher 会负责处理跨主机的联网，以及安排容器到合适的主机上运行。图 12-5 展示了在一个由 Rancher 管理的拥有两台主机的集群上运行的 identidock。

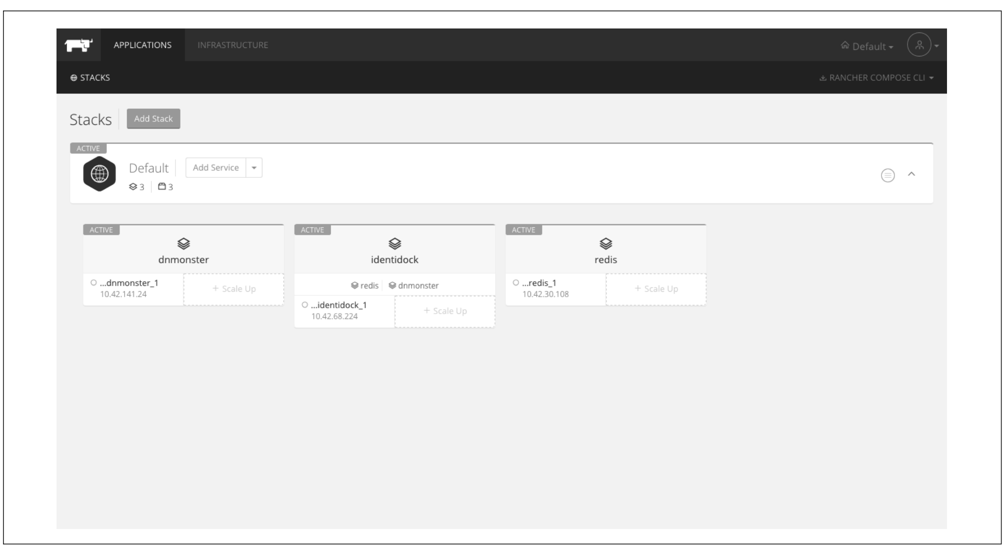


图 12-5：在 Rancher 上运行 identidock

另外，也可以使用 Rancher Compose（https://github.com/rancher/rancher-compose），它是一个命令行工具，用于把 Docker Compose 文件部署到 Rancher 之上。

使用 Rancher 后，你就能很轻松地获取运行中的容器的信息，查看它们的日志，甚至可以运行 shell 对进程进行调试。

虽然目前 Rancher 以自己的方式来实现跨主机联网（利用 IPsec，https://en.wikipedia.org/wiki/IPsec）、服务发现和简单的服务编排，但它正打算日后全盘改用 Docker 的栈（也就是说，服务编排将使用 Swarm，联网将采用 Docker 全新的插件框架）。它也正在努力与 Kubernetes 和 Mesos 进行整合。Rancher 非常容易上手，在现有的系统上应用它也是轻而易举，因此很值得一试。

12.2.2 Clocker

Clocker（http://www.clocker.io/）是一套基于 Apache Brooklyn（http://brooklyn.apache.org/）的自托管开源容器管理平台。与 Rancher 相比，它提供的方案更偏向于应用，而且它还允许在一个应用程序中混合使用虚拟机和容器。通过 jclouds（https://jclouds.apache.org/）工具包，它能够支持非常多的云服务平台。

掌握 Clocker 需要下一些工夫。下载之后，配置时你需要提供用于部署云服务的令牌以及访问密钥。配置完成后，便可以旋即启动 Clocker 云了，之后它会自动部署主机，还会自动安装 Weave 或 Project Calico 供联网之用。

一旦 Clocker 云准备就绪，便可以启动一个用于运行 Docker 容器的全新应用。以下的 YAML 文件可以在 Clocker 上启动 identidock：

```
id: identidock
name: "Identidock"
location: my-docker-cloud
services:
- type: docker:redis:3
  name: redis
  id: redis
  openPorts:
  - 6379
- type: docker:amouat/dnmonster:1.0
  name: dnmonster
  id: dnmonster
  openPorts:
  - 8080
- type: docker:amouat/identidock:1.0
  name: identidock
  id: identidock
  portBindings:
    80: 9090
  links:
  - $brooklyn:component("redis")
  - $brooklyn:component("dnmonster")
```

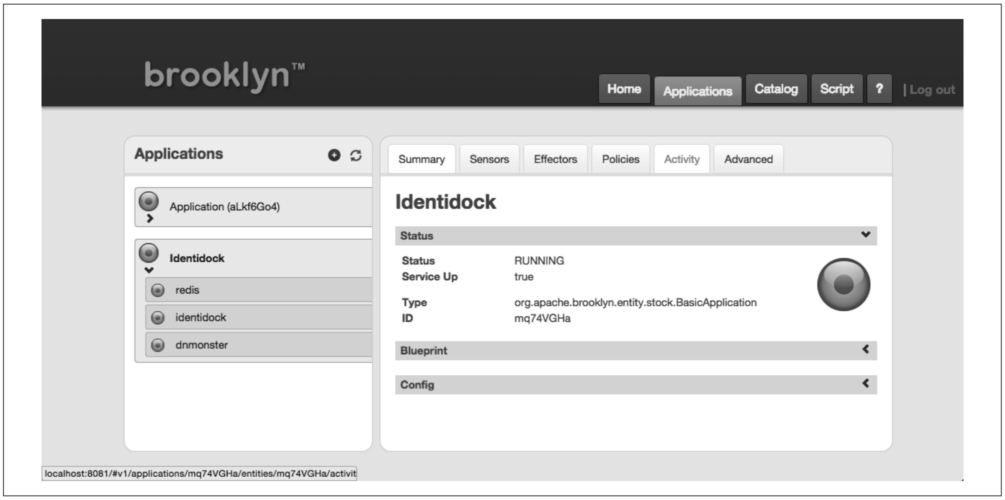


图 12-6：Clocker 正利用 Weave 和 AWS 运行 identidock

Clocker 的优点在于，我们可以很轻松地将一个 Docker 应用与非容器化的资源混合使用。例如，通过下面这个 Redis 服务的 YAML 文件，我们可以把 Redis 容器换成一个 Redis 虚拟机：

```
- type: org.apache.brooklyn.entity.nosql.redis.RedisStore
  name: redis
  location: jclouds:softlayer:lon02
  id: redis
  install.version: 3.0.0
  start.timeout: 10m
```

这个配置文件的意思是，在 SoftLayer 的伦敦数据中心部署一个虚拟机，并在虚拟机内安装和运行 Redis。

撰写这本书的时候，Clocker 正在快速发展。尽管它目前与其他解决方案相比显得有点粗糙，但它选择了使用 Brooklyn 和 jclouds 的技术，这就意味着它可以部署在很多不同的系统上，而且对于混合各种不同类型的基础设施，它也能应付自如。

12.2.3　Tutum

Tutum（https://www.tutum.co/）[6] 提供了一个用于部署和管理容器的托管平台。Tutum 非常注重可用性，因此它的界面设计非常简洁。

Tutum 添加节点的方式有两种，一种是通过提供公有云的用户凭据，另一种是在现有的机器上安装 Tutum 代理。代理以守护进程的形式运行，而不是一个容器，这意味着它不能支持所有操作系统（尤其是不支持由 Docker Machine 创建的 boot2docker 镜像）。

注 6：Docker 于 2015 年 10 月宣布收购 Tutum，该网站已于 2016 年 5 月 31 日停止运作，其服务已合并到 Docker 的 Docker Cloud（https://cloud.docker.com/）服务。——译者注

对于需要互联的服务，Tutum 使用 stackfiles 来定义不同的服务群组。它刻意与 Compose 文件设计得非常相似，但添加了一些与服务编排和扩展相关的额外字段，如 `target_num_containers` 和 `deployment_strategy`，同时摒弃了诸如 `user` 和 `cap_add` 等字段。

Tutum 的内部采用 Weave（参见 11.5.2 节）作为跨主机网络和服务发现的工具。

利用 Tutum 把 identidock 运行起来也很轻松。图 12-7 展示的是 Tutum 的仪表盘，其中显示 identidock 正运行在两个 Digital Ocean 的节点上。

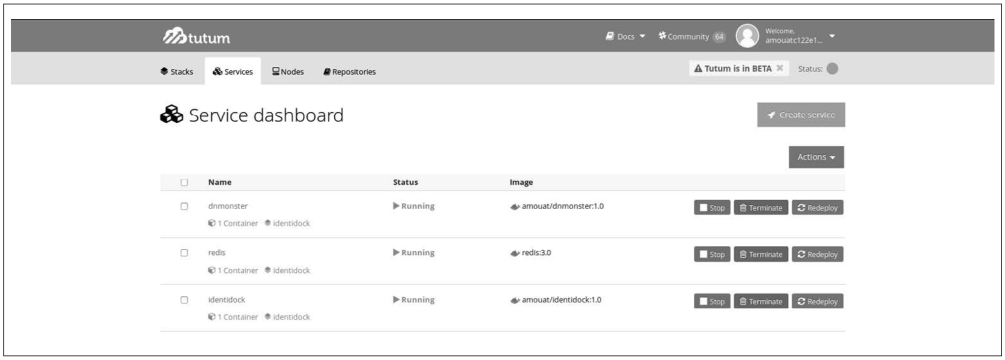


图 12-7：在 Tutum 上运行 identidock

除了 Web 界面外，Tutum 还可以通过 REST API 和 Tutum 的命令行工具进行访问。

如果有任何人需要一个集中式的服务来帮助减轻大部分的运维工作，包括建立、配置和运行容器化服务，那么 Tutum 很值得一试。如果你希望对服务有绝对的控制权，或者你对信任一个集中式的服务抱有疑虑，那么 Tutum 恐怕就不太适合你了。

12.3 总结

显然，有很多服务编排、集群以及容器管理的方案可供选择。然而，不同的选择之间的分野一般都很明确。就服务编排而言，我的观点如下。

- Swarm 的优点（同时也是它的缺点）在于它使用了标准的 Docker 接口。这使得无论是使用它还是把它整合到现有的工作流程都非常容易。但另一方面，一些更复杂的调度方法可能需要用到自定义接口，支持它们就会比较困难。
- fleet 是一个低阶并且相当简单的编排层，可以作为运行更高阶的编排工具（如 Kubernetes）或定制系统的基础。
- Kubernetes 作为一个编排工具，它自带服务发现和复制的功能，并且很坚持自己的一套设计理念。使用它时或许需要对现有应用程序重新设计，但如果使用得当，你将拥有一个可容错和可扩展的系统。
- Mesos 是一个低阶而且经过千锤百炼的调度程序，支持多个容器编排框架，包括 Marathon、Kubernetes 和 Swarm。

本书写作之际，Kubernetes 和 Mesos 比 Swarm 更成熟也更稳定。扩展性方面，只有 Mesos 被证实能够支持几百个到几千个节点的大型系统。然而，对于小型的集群，比如顶多十几个节点的集群来说，Mesos 或许会过于繁复。

管理平台方面，对于单纯的 Docker 部署，Rancher 是个不错的选择。它可以很方便地添加到现有的系统或从系统移除，因此可以放心试用。

第13章

容器安全与限制容器

为了能够安全地使用 Docker，你必须对潜在的安全问题有清楚的认识，并且对保护容器化系统的主要工具和技术有所了解。本章将从生产环境中运行 Docker 的角度出发来讨论其安全性，不过大部分建议同样适用于开发环境。即使增加了安全性的考虑，保持开发环境和生产环境的一致性也仍然非常重要，以避免代码在不同的 Docker环境之间迁移时出现问题。

网上关于 Docker 的帖子和新闻报道[1]可能给你留下这样的印象：Docker 本质上并不安全，它不适合用于生产环境。诚然，你必须对如何安全使用容器有所了解，但如果容器使用得当，它将能带来比虚拟机或裸机更安全和更高效的系统。

本章的开始将探讨一些围绕容器化系统安全展开的事项，这些事项在你使用容器时必须考虑进去。

免责声明

本章中的指导仅代表我个人的意见。我不是安全研究员，也没有负责运行任何面向公众的大型系统。话虽如此，我相信任何遵循本章指导的系统，将比现今大部分的系统更为安全。本章的建议并不构成一套完整的解决方案，只能作为你开发自己的安全措施与策略时的依据。

注 1：有关 Docker 安全性方面比较好的文章，包括由红帽公司的 Dan Walsh 在 opensource.com 网站上刊登的系列文章（https://opensource.com/business/14/7/docker-security-selinux），以及 Jonathan Rudenberg 关于镜像的安全隐患的文章（https://titanous.com/posts/docker-insecurity），但请注意，在 Jonathan 的文章中提及的问题，很多都已经通过摘要（digest）和公证服务（Notary）项目得到解决。

13.1 需要考虑的事项

那么在一个容器化的环境中，有哪些安全事项是你必须考虑的呢？下面的列表并不完全，但应该足以启发一些思考。

内核漏洞

与虚拟机不同，所有容器以及主机都共同使用一个内核，因此内核漏洞的严重性将会被放大。如果一个容器引起内核崩溃，那么整个主机都会被弄垮。虚拟机的情况就好很多：攻击者必须先经过虚拟机内核和虚拟机管理程序，才能够接触到主机内核。

拒绝服务（DoS）攻击

所有容器共同使用相同的内核资源。因此，如果一个容器可以独占某些资源的访问权，包括内存和一般不太受人关注的资源，例如用户 ID（UID），那么将会导致主机上的其他容器资源枯竭，从而拒绝服务，使合法用户无法访问部分或整个系统。

容器突破

即使攻击者能够获得一个容器的访问权限，他也不能进一步获得其他容器或主机的访问权。由于 Docker 的用户管理并未采用命名空间[2]，任何可以突破容器的进程在主机上将获得与容器内相同的权限。如果你在容器内是 root，你在主机上同样是 root。[3] 这也意味着，你必须留心潜在的*权限提升*（privilege escalation）攻击，这种攻击往往是通过一些需要额外权限的应用程序中的错误，使普通用户能够获取更高级别的权限，例如 root 用户的权限。由于容器技术仍处于起步阶段，你在考虑安全性的时候，必须假设容器突破出现的可能性虽然较低，但并非完全不可能。

镜像污染

你怎么知道你正在使用的镜像是安全的呢？它们有没有被篡改过？镜像真的是来自它们所声称的来源吗？如果攻击者可以欺骗你运行他的镜像，那么你的主机和数据将受到威胁。同样，你必须确保你运行的镜像是最新的，并保证不包含带有已知漏洞的软件版本。

密钥泄露

当容器访问数据库或某服务时，一般需要使用某种形式的密钥，例如 API 密钥，或用户名和密码。攻击者如果能够掌握密钥，他就能够访问这些服务。由于微服务架构中的容器不断重复启动和停止，相较于只有数个长时间运行的虚拟机的架构，它的问题就更为严重。之前的 9.7 节中已经讨论过共享密钥的解决方案。

注 2：Docker 已从 1.10 版本开始正式支持用户命名空间。——译者注

注 3：目前 Docker 正在开发把容器中的 root 用户映射到主机上的普通用户的功能。这将大大削弱当容器突破发生时攻击者的能力，但会产生新的数据卷所有者问题。

容器与命名空间

在一篇广泛引用的文章中，红帽公司的 Dan Walsh 提出了“Containers Do Not Contain”（https://opensource.com/business/14/7/docker-security-selinux）的看法。

他主要表达的意思是，容器使用的资源并非全部采用了**命名空间**。**已采用**命名空间的资源在主机上会被映射到其他的值（例如，容器中 PID 1 的进程，在主机上或其他容器中它们的 PID 并不是 1）。与此相反，未采用命名空间的资源在主机和容器上都是一样的。

未采用命名空间的资源列举如下。

用户 ID（UID）

容器内用户的 UID 和在主机上是相同的。这就意味着，如果容器以 root 用户运行时遭受攻击，那么攻击者就能拿到主机上的 root 身份。目前，把容器的 root 用户映射到主机上的一个高 ID 用户的开发工作正在进行中，但目前尚未完成。

内核钥匙圈

如果你的应用或相关的程序使用内核钥匙圈来处理密钥或类似的东西，那么你**必须**清楚这一点：密钥是按 UID 区分的，因此使用相同 UID 运行的容器，以及主机上的相同用户，都可以取得相同的密钥。

内核本身以及任何内核模块

当容器加载内核模块时（这需要额外权限），主机和所有容器都能够使用这个模块，包括稍后讨论的 Linux 安全模块（Linux Security Modules）。

设备

包括磁盘驱动器、声卡和图形处理单元（GPU）。

系统时间

如果在容器内更改时间，那么主机和所有容器的系统时间都会被一并更改。只有被授予 `SYS_TIME` 能力（capability）的容器才可以这样做，而且这不是默认准许的。

毋容置疑，无论是 Docker 还是它依赖的底层 Linux 内核技术都尚未成熟，远不如同等的虚拟机技术那么久经考验。至少目前而言，容器尚未提供与虚拟机相同水平的安全保证。[4]

注 4：有一个有趣的争论，争论的焦点是容器能否像虚拟机一样安全。虚拟机的支持者认为，缺乏虚拟机管理程序以及需要共享内核资源，意味着容器的安全性永远不会太高。容器的支持者则认为，虚拟机由于其攻击面较大而更容易被入侵，并指出虚拟机需要大量复杂且高权限的代码，用于模拟不常用的硬件 [近期的一个例子就是 VENOM（http://venom.crowdstrike.com/）漏洞，它利用了模拟软盘驱动器的代码中存在的漏洞]。

13.2 纵深防御

那么你能做些什么呢？你可以假设漏洞存在，用纵深防御策略保护自己。试想一座城堡，里面设有多重防御，皆是为阻止各类攻击而设计的。通常，城堡要利用护城河或当地的地理环境来控制进出的路径。城墙用很厚的石头建造，以抵御火攻和炮弹的攻击。城墙内还有用作防御的城垛和多层的主楼。即使侵略者越过了一道防线，在他面前还有更多的难关。

你的系统防御策略也应该是多重的。例如，容器可以置于虚拟机中运行，假如容器被突破了，还有另一重防御可以防止攻击者到达主机或属于其他用户的容器。监控系统必须就位，确保在侦测到异常行为时通知管理员。你还应该部署防火墙，限制从网络访问容器的权限，以缩小来自外部的攻击面。

最小权限

另一个必须坚持的重要原则就是*最小权限*。每个进程和容器，应以足够执行其功能的最低访问权限和资源运行。[5] 这种方法的主要好处是，当一个容器被入侵后，攻击者尝试访问其他数据或资源时，仍然会受到严重限制。

- 你可以采取如下的许多措施来限制容器的能力，以达到最低权限的目的。
- 确保容器内的进程没有使用 root 身份运行，这样即使进程的安全漏洞被利用，攻击者仍然无法取得 root 权限。
- 把文件系统设为只读，使攻击者无法覆写数据以及写入恶意脚本。
- 减少容器能够调用的内核函数，以缩小潜在的攻击面。
- 为避免已被入侵的容器或程序消耗过多资源（如内存或 CPU）以至于把主机拖垮，从而导致拒绝服务（DoS）攻击，可以限制容器允许使用的资源数量。

> Docker 权限 == Root 权限
>
> 本章重点介绍运行容器时需要注意的安全问题，不过你也必须注意谁有访问 Docker 守护进程的权限，这一点非常重要。任何能够启动和运行 Docker 容器的人，都相当于有了主机的 root 权限。举个例子，假设你能够运行这个命令：
>
> ```
> $ docker run -v /:/homeroot -it debian bash
> ...
> ```
>
> 你现在就可以访问主机上的任何文件或程序了。
>
> 如果你的 Docker 守护进程允许远程 API 访问，请务必注意如何保证它的安全，以及什么人可以访问它。如果可能的话，限制它只能在内网中访问。

注 5：最小权限的概念首先由 Jerome Saltzer 在“Protection and the control of information sharing in multics”一文中提出，他在其中说道：“系统上的每个程序和每个特权用户，执行时应当使用最低程度的权限，足够完成任务便可。”（*Communications of the ACM*, vol. 17）近日，Docker 的 Diogo Mónica 和 Nathan McCauley 基于 Saltzer 发表的原则，提倡“最小权限的微服务”这一想法。

13.3 如何保护identidock

为了使读者更好地了解这一章的内容，这一节会介绍如何保护 identidock 在生产环境时的安全。identidock 没有存储任何敏感信息，因此我们主要关心的是，攻击者入侵服务器后把它变成发送垃圾信息或类似的工具。我假设该 identidock 具有一定用户基础，如果它停止工作的话，至少它的用户群会感到不满。[6]

需要确保的主要事项如下。

- 在虚拟机或专用主机上运行 identidock 容器，以免其他用户或服务受到攻击（参见 13.4 节）。
- 负载均衡器或反向代理是唯一把端口暴露于外界的容器，这将使得攻击面大幅度缩减。任何监控或日志记录服务只通过私有接口或 VPN 开放（参见 13.7.2 节）。
- 所有 identidock 镜像必须定义一个普通用户，并且不会以 root 身份运行（参见 13.7.1 节）。
- 所有 identidock 镜像必须以散列值下载，或以安全和可验证的方式获得（参见 13.6 节）。
- 确保用于检测异常流量或行为的监控和警报通知机制已各就各位（参见第 10 章）。
- 所有容器运行的软件都是最新的，而且以生产模式运行，调试信息必须关闭（参见 11.5 节）。
- 如果主机已有 AppArmor 或 SELinux，那么就使用它们（参阅 13.9.1 节和 13.9.2 节）。
- 对 Redis 增加某种形式的访问控制或密码保护。

如果还有足够的时间，我会采取以下措施。

- 从 identidock 镜像删除所有不必要的 setuid 二进制文件。一旦攻击者获得容器的访问权，他就有可能进一步提权，这样做可以减少这一风险（参见 13.7.3 节）。
- 在可能的情况下，把文件系统设置为只读。dnmonster、identidock 和 redis 容器可以在只读的容器文件系统运行，但 redis 的数据卷必须为可写入（参见 13.7.7 节）。
- 放弃不必要的内核权限。dnmonster 和 identidock 容器运行时将放弃全部内核能力（参见 13.7.8 节）。

如果我仍有疑虑，或者运行着一个安全敏感度很高的服务，我会采取以下措施。

- 以 `-m` 参数限制每个容器的内存上限。这样做可以防止一些 DoS 攻击和内存泄漏。前提是需要对容器进行分析，以确定最大的内存使用量，否则只能使用一个非常宽松的限制值。
- 以专门类型的容器运行 SELinux。这是一个非常有效的安全措施，但使用它需要做大量工作，而且必须使用 devicemapper 存储驱动程序（参见 13.9.1 节）。
- 通过 `ulimit` 限制进程数量。这个限制是针对容器用户生效的，因此它比想象中要难用。它有助于阻止用作 DoS 攻击的 fork 炸弹的威胁（参见 13.7.9 节）。
- 加密内部通信，提高攻击者篡改数据的难度。

此外，系统应该有定期审核，以确保一切都是最新的，以及容器没有过度占用资源。即使是像 identidock 这样的小应用，也有很多安全措施，而且还有更多的措施可以考虑。

本章的其余部分将详细讲述如何实施这些防御措施。有一个要点必须牢记，那就是检查和

注 6：最好不要袖手旁观……

关卡越多，就越有机会阻止攻击者造成真正的破坏。若防御措施使用不当，则可能会开放新的攻击向量，从而影响 Docker 的系统安全性。若使用得当，则能在提升系统隔离性的同时限制应用可能对系统造成的破坏，从而最终提升系统安全性。

13.4 以主机隔离容器

如果你的配置是为多个容器用户提供服务的多租户架构（无论是机构内部用户，还是外部客户），你必须确保每个用户都被安置在一个独立的 Docker 主机上，如图 13-1 所示。相比于多个用户共用主机的做法，这样做效率较低，并且将会增加虚拟机或机器的数量，但对于安全性而言却是非常重要的。这样做最主要的原因是防止容器突破，容器突破会导致攻击者能够进入其他用户的容器，或取得其他用户的数据。假如容器真的被突破了，攻击者也会被限制在该虚拟机或机器内，而无法轻松地访问属于其他用户的容器。

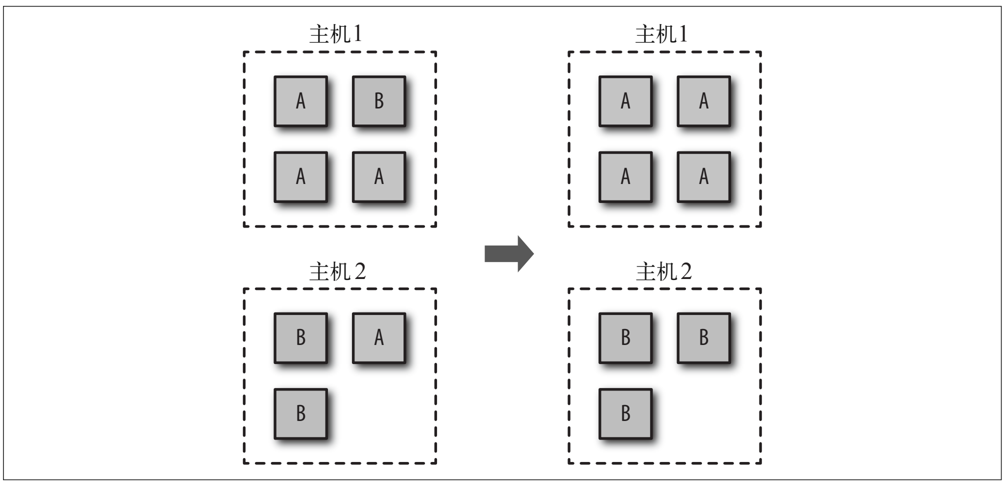


图 13-1：以主机隔离容器

同样，如果你的容器负责处理或储存敏感信息，那么你必须将它们放在一个单独的主机上，使其与处理敏感度较低的信息的容器分离，尤其是那些运行的直接面向最终用户的应用程序的容器。例如，处理信用卡信息的容器必须与运行 Node.js 的前端容器分离。

分离和使用虚拟机还可以提供对 DoS 攻击的额外保护。若用户只能在自己的虚拟机中运行，则独占主机内存以致其他用户无法获取所需资源的情况将不可能发生。

在中短期内，绝大多数容器的部署将涉及虚拟机的使用。尽管这并不理想，但确实意味着我们可以兼得容器的高效性与虚拟机的安全性。

13.5 进行更新

能够迅速更新运行中的系统，对于维持系统的安全性至关重要，尤其是当漏洞出现在常用的工具或框架中时。

更新容器化系统的过程大致包括以下步骤。

(1) 识别出需要更新的镜像，包括基础镜像和任何相关镜像。下面的辅助栏“获取运行中镜像的列表”将解释如何利用命令行工具做到这一点。
(2) 获取或创建每个基本镜像的更新版本。把这个新版本上传到寄存服务器或下载站点。
(3) 对于每一个相关的镜像，执行 `docker build` 并加上 `--no-cache` 参数，然后再把镜像上传。
(4) 在每个 Docker 主机上，执行 `docker pull` 以确保所有主机的镜像都是最新的。
(5) 重启每个 Docker 主机上的容器。
(6) 当确定一切工作正常时，你可以删除主机上旧的镜像。如果可以的话，也把寄存服务器中的镜像删除。

其中的一些步骤说起来容易，但做起来并不是那么回事。确定需要更新的镜像，可能需要执行一些单调乏味的工作和 shell 技巧。重启容器这一操作假设你已经实现了某种对滚动更新的支持，或者你可以忍受停机时间。在撰写本书的时候，尚不能完全删除寄存服务器中的镜像并取回被镜像占用的磁盘空间，只能等待这个功能完成开发。[7]

如果你使用 Docker Hub 来构建镜像，可以利用仓库链接把你的镜像与其他镜像相连，当被连接的镜像发生变化时，你的镜像随即重新构建。因此，通过与基础镜像相连，当基础镜像发生变化时，你的镜像就会自动重建。

获取运行中镜像的列表

下面的命令能够查看所有运行中镜像的 ID：

```
$ docker inspect -f "{{.Image}}" $(docker ps -q)
42a3cf88f3f0cce2b4bfb2ed714eec5ee937525b4c7e0a0f70daff18c3f2ee92
41b730702607edf9b07c6098f0b704ff59c5d4361245e468c0d551f50eae6f84
```

多加一些 shell 命令，可以获取更多信息：

```
$ docker images --no-trunc | \
    grep $(docker inspect -f "-e {{.Image}}" $(docker ps -q))
nginx   latest  42a3cf88f...  2 weeks ago  132.8 MB
debian  latest  41b730702...  2 weeks ago  125.1 MB
```

下面的命令能够获取所有镜像，以及它们的基础或中间镜像（要取得完整的 ID，可以加上 `--no-trunc` 参数）：

```
$ docker inspect -f "{{.Image}}" $(docker ps -q) | \
    xargs -L 1 docker history -q
41b730702607
3cb35ae859e7
42a3cf88f3f0
e59ba510498b
50c46b6286b9
ee8776c93fde
439e7909f795
```

注 7：暂时的解决方法是，把所有要保留的镜像下载下来，然后把它们推送到一个全新、干净的寄存服务器。

```
0b5e8be9b692
e7e840eed70b
7ed37354d38d
55516e2f2530
97d05af69c46
41b730702607
3cb35ae859e7
```

用同样的方法，可以扩展刚才的命令，获得这些镜像的详细信息：

```
$ docker images | \
   grep $(docker inspect -f "{{.Image}}" $(docker ps -q) | \
          xargs -L 1 docker history -q | sed "s/^/\-e /")
nginx   latest  42a3cf88f3f0  2 weeks ago  132.8 MB
debian  latest  41b730702607  2 weeks ago  125.1 MB
```

如果你希望结果中还包括中间镜像以及已命名的镜像的详细信息，那么需要在 docker images 命令中加上 -a 参数。请注意，这个命令有一个很容易出错的地方，那就是如果主机上没有基础镜像的标签版本，它是不会出现在列表中的。例如，官方的 Redis 镜像是基于 debian:wheezy 的，但如果你还没有单独在主机上明确地把 debian:wheezy 镜像下载下来（而且镜像版本必须完全一致），它在 docker images -a 的输出中将显示为 <None>。

当你需要修复一个在第三方镜像中出现的漏洞（包括官方镜像）时，你只能依靠镜像提供方及时发布更新。在过去，供应商曾被批评反应迟缓。如果遇到这种情况，你要么等待他们发布更新，要么自己来构建镜像。假设你拥有镜像的 Dockerfile 和源代码，构建自己的镜像可能是一个简单而有效的临时解决办法。

这种做法可以与典型的虚拟机做法作比较。一般虚拟机采用诸如 Puppet、Chef 或 Ansible 等配置管理（configuration-management，CM）软件，虚拟机并不会重建，只会在需要的时候通过 SSH 命令或虚拟机中的代理程序进行更新和修补。虽然这种方法可行，但会造成不同虚拟机的状态不一，跟踪和更新虚拟机也变得异常复杂。然而必须这样做，因为可以避免重新创建虚拟机，并免去维护一个源镜像或黄金镜像的开销。虽然容器也可以采用配置管理的方法，但这样做将导致复杂程度大增，并且得不到任何好处。由于容器启动速度快，镜像易于构建和维护，采用黄金镜像的方法将较为简单，而且效果良好。[8]

给镜像加标签

构建镜像时尽量使用标签，这有助于识别镜像以及镜像所包含的内容。这个功能在 1.6 版本中出现，允许镜像的作者把镜像关联任意的键和值。在 Dockerfile 中可以做到：

```
FROM debian
LABEL version 1.0.
LABEL description "A test image for describing labels"
```

注 8：这种做法与现今的“基础设施不可改变”这一概念非常相似，它的意思是，基础设施（包括裸机、虚拟机和容器）是绝对不会被修改的，如果有修改的必要，那么它们只能被替换。

你还可以更进一步，添加如 git 散列值等数据，用于标示构建镜像时编译的代码版本，但这需要使用模板工具或类似方法来自动更新这个值。

标签也可以在容器运行时附加给它：

```
$ docker run -d --name label-test \
                    -l group=a debian sleep 100
1d8d8b622ec86068dfa5cf251cbaca7540b7eaa67664a13c620006...
$ docker inspect -f '{{json .Config.Labels}}' label-test
{"group":"a"}
```

当你希望在运行时处理某些事件，例如把容器动态分配至负载均衡器时，这将非常有用。

有时候为了获得 Docker 的新功能、安全补丁或问题修复，需要更新 Docker 的守护进程。这时只能把所有容器迁移到一台新的主机，或者在进行更新时暂时把容器停止。建议订阅 docker-user（https://groups.google.com/forum/#!forum/docker-user）或 docker-dev（https://groups.google.com/forum/#!forum/docker-dev）谷歌群组，以便获得重要更新的通知。

避免使用不再支持的驱动程序

尽管 Docker 出现的时间不长，它已经历了好几个发展阶段，甚至有些功能已经过时或不再维护。继续依赖这些功能将产生安全隐患，因为它们不像 Docker 的其他部分一样受到同等程度的关注和更新。同样情况也适用于与 Docker 关联的驱动和扩展程序。

在此要特别指出的是，LXC 的执行驱动已经过时了，你不应该使用它。现在它是默认关闭的，但你应该检查你的守护进程，确保运行时没有使用 `-e lxc` 参数。

存储驱动程序是另一个开发非常活跃且更改频繁的功能。本书写作之际，Docker 推荐的存储驱动程序正从 AUFS 转向 Overlay。AUFS 已被移出内核，并且不再开发，因此建议 AUFS 的用户尽早转向使用 Overlay 驱动。

13.6 镜像出处

为了能安全地使用镜像，我们需要保证镜像的*出处*（provenance）的真实性，即它们来自哪里，以及由谁创建。必须确保我们得到的镜像与原作者测试的是同一个，无论是在储存还是传送的过程中都没有被篡改过。如果无法验证这一点，那么镜像有可能已经损坏了；而更糟糕的是，它还有可能被换成含有恶意程序的镜像。前文已经讨论过 Docker 的安全问题，这确实是一个关注的重点，你应该假设恶意镜像真的能够完全控制你的主机。

在计算机领域，出处并非一个新出现的问题。建立软件或数据出处的主要工具是*安全散列*。安全散列类似于数据指纹，它是一个（相对而言）长度很短的字符串，针对不同数据能产生独一无二的散列值。任何对数据的改变将导致散列值发生变化。计算安全散列有多个不同的算法，区别在于复杂度和散列值唯一性的保证。最常见的算法是 SHA（它有数个版本）和 MD5（它本身存在一些问题，应避免使用）。如果拥有数据的安全散列和数据本身，我们便能够利用数据重新算出散列值，然后把两个散列值进行比较。如果两个散列值

一样，那么可以肯定数据没有损坏或被篡改。然而，你可能会问：为什么我们可以相信散列值？攻击者难道不会同时修改数据和散列值吗？回答这个问题的最佳答案是*加密签名*和公共 / 私有密钥对。

通过加密签名，我们可以验证一个物件的发布者的身份。如果发布者使用他的*私钥*[9] 签署了一个物件，那么这个物件的任何接收者可以利用发布者的*公钥*验证该签名，以核实这个物件是否来自这个发布者。假如接收一方已经取得发布者的公钥，而发布者的密钥没有被攻破或泄露，那么我们可以肯定内容确实来自该发布者，而且数据没有被篡改过。

13.6.1 Docker摘要

在 Docker 中，安全散列被称为*摘要*（digest）。摘要是一个文件系统层或清单（manifest）的 SHA256 散列值，清单是一个元数据文件，用来描述 Docker 镜像的组成部分。由于清单中包含一个所有镜像层的列表，而当中都附上每个镜像层的摘要[10]，因此如果你能够验证清单没有被篡改，那就可以放心下载和信任所有镜像层，即使是通过不可靠的通道（例如 HTTP）。

13.6.2 Docker的内容信任机制

Docker 在 1.8 版本中引入了内容信任机制（content trust），开发者[11] 通过它能够对他们的内容进行签署。这个机制出现后，Docker 的内容发布机制已经是完全可信了。当用户从仓库下载镜像时，他同时会收到一张包含发布者公钥的证书，让他可以验证镜像是否真正来自该发布者。

当内容信任机制启用后，Docker 引擎只会使用经过签署的镜像，并拒绝运行任何签名或摘要不匹配的镜像。

下面来看看内容信任机制的实际使用情况。首先尝试下载已签署和未签署的镜像：

```
$ export DOCKER_CONTENT_TRUST=1  ❶
$ docker pull debian:wheezy
Pull (1 of 1): debian:wheezy@sha256:c584131da2ac1948aa3e66468a4424b6aea2f33a...
sha256:c584131da2ac1948aa3e66468a4424b6aea2f33acba7cec0b631bdb56254c4fe: Pul...
4c8cbfd2973e: Pull complete
60c52dbe9d91: Pull complete
Digest: sha256:c584131da2ac1948aa3e66468a4424b6aea2f33acba7cec0b631bdb56254c4fe
Status: Downloaded newer image for debian@sha256:c584131da2ac1948aa3e66468a4...
Tagging debian@sha256:c584131da2ac1948aa3e66468a4424b6aea2f33acba7cec0b631bd...
$ docker pull amouat/identidock:unsigned
No trust data for unsigned
```

❶ 在 Docker 1.8 中，必须设置环境变量 `DOCKER_CONTENT_TRUST=1` 才能启用内容信任机制。而往后的版本中，这会是默认启用的。

注 9：公开密钥加密（public-key cryptography）的内容很吸引人，但关于它的完整讨论超出本书的范围。详情请参阅 Bruce Schneier 的《应用密码学》一书。

注 10：类似的结构被应用在诸如 BitTorrent 的协议和比特币中，被称为*散列表*（hash list）。

注 11：这一章中提到的所谓发布者，指的是推送镜像的任何人，并不仅限于大企业或机构。

可以看到，由官方签署的 Debian 镜像可以成功下载。然而，Docker 拒绝下载未签署的 `amouat/identidock:unsigned` 镜像。

那么，如何推送已签署的镜像呢？做起来极其容易：

```
$ docker push amouat/identidock:newest
The push refers to a repository [docker.io/amouat/identidock] (len: 1)
...
843e2bded498: Image already exists
newest: digest: sha256:1a0c4d72c5d52094fd246ec03d6b6ac43836440796da1043b6ed8...
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new offline key with id 70878f1:
Repeat passphrase for new offline key with id 70878f1:
Enter passphrase for new tagging key with id docker.io/amouat/identidock ...
Repeat passphrase for new tagging key with id docker.io/amouat/identidock ...
Finished initializing "docker.io/amouat/identidock"
```

由于这是启用内容信任机制后第一次推送镜像到这个仓库，Docker 创建了一个新的*根签名密钥*（root signing key）和*标记密钥*（tagging key）。稍后会再回来解释什么是标记密钥，但现在请谨记，你必须保持根密钥的安全性和保密性，这一点非常重要。如果遗失了它，那么会非常麻烦，你的仓库的用户必须手动把旧的证书删除，否则他们将无法下载新的镜像或更新现有镜像。

现在，可以在内容信任机制中下载我们的镜像：

```
$ docker rmi amouat/identidock:newest
Untagged: amouat/identidock:newest
$ docker pull amouat/identidock:newest
Pull (1 of 1): amouat/identidock:newest@sha256:1a0c4d72c5d52094fd246ec03d6b6...
sha256:1a0c4d72c5d52094fd246ec03d6b6ac43836440796da1043b6ed81ea4167eb71: Pul...
...
7e7d073d42e9: Already exists
Digest: sha256:1a0c4d72c5d52094fd246ec03d6b6ac43836440796da1043b6ed81ea4167eb71
Status: Downloaded newer image for amouat/identidock@sha256:1a0c4d72c5d52094...
Tagging amouat/identidock@sha256:1a0c4d72c5d52094fd246ec03d6b6ac43836440796d...
```

如果你从来没有在这个仓库下载过镜像，Docker 会首先取得这个仓库的发布者证书。这个动作通过 HTTPS 完成，因此风险很低。与首次使用 SSH 连接至主机时的情况相似，如果 Docker 显示的凭据正确，你需要表示对它信任。以后再次从这个仓库下载镜像时，镜像便会以现有的证书进行验证。

备份签名密钥

Docker 将加密所有不在使用中的密钥，并确保私密信息不会写入磁盘。由于密钥的重要性，我建议把它们备份在两个加密的 U 盘上，然后把 U 盘存放在安全的地方。下面的命令可用于创建密钥的 TAR 文件：

```
$ umask 077
$ tar -zcvf private_keys_backup.tar.gz \
            ~/.docker/trust/private
$ umask 022
```

umask 命令确保文件权限为只读。

需要注意的是，根密钥只用于创建或撤消密钥，因此它可以在不使用时离线存储，为了安全起见你也应该这样做。

现在回到刚才提到的标记密钥。每个仓库都会生成一个标记密钥，并为发布者所拥有。标记密钥由根密钥签署，这使得任何用户都能够通过发布者的证书来验证它。标记密钥可以在一个组织内共享，用于签署这个仓库中的任何镜像。标记密钥产生后，根密钥便可以拿到安全的地方离线存储，你也应该这样做。

万一标记密钥泄露了，它还是有可能恢复的。通过轮转标签密钥，已泄露的密钥便可以从系统中删除。这个过程是在用户不知情的情况下进行的，而且你可以预先采取行动，以防未知的密钥泄露。

内容信任机制还可以保证镜像为最新，从而能够抵御*重放攻击*（replay attack）。重放攻击缘于某事物被替换成先前有效时的状态。例如，攻击者把一个二进制文件替换成先前由发布者签署，但含有安全漏洞的旧版本。由于这个二进制文件的签名是正确的，用户便有可能被诱骗运行这个含有漏洞的版本。为了避免这种事情发生，内容信任机制利用每个仓库都有的*时间戳密钥*。时间戳密钥用于签署与仓库相关的元数据。元数据的有效期很短，因此需要时间戳密钥不断地为它重新签署。在下载镜像前，先验证元数据是否过期，Docker 客户端便可以确认它接收的镜像是最新的。时间戳密钥由 Docker Hub 管理，无需发布者做任何管理工作。

一个仓库可以同时存储已签署和未签署的镜像。如果你已启用内容信任机制，但希望下载未签署的镜像，可以使用 --disable-content-trust 参数：

```
$ docker pull amouat/identidock:unsigned
No trust data for unsigned
$ docker pull --disable-content-trust amouat/identidock:unsigned
unsigned: Pulling from amouat/identidock
...
7e7d073d42e9: Already exists
Digest: sha256:ea9143ea9952ca27bfd618ce718501d97180dbf1b5857ff33467dfdae08f57be
Status: Downloaded newer image for amouat/identidock:unsigned
```

想要了解更多有关内容信任机制（https://docs.docker.com/engine/security/trust/content_trust/）的内容，请参阅 Docker 的官方文档以及更新框架规范（The Update Framework，https://theupdateframework.github.io/），内容信任是基于这个规范实现的。

虽然这个机制颇为复杂，其中还涉及多套密钥，但 Docker 一直致力于确保最终用户能够轻松地使用它。凭借内容信任机制，Docker 已开发出一个用户友好和现代化的安全框架，以保证镜像出处的真实性、镜像一直为最新，以及镜像的完整性。

内容信任机制在 Docker Hub 上已经启用，如今正在运作中。如果你需要在本地的寄存服

务中配置内容信任机制，那么还需要配置和部署公证服务器（Notary server，https://github.com/docker/notary）。

公证服务

Docker 公证项目（https://github.com/docker/notary）是一个通用的 C/S 架构服务，它允许在发布和访问内容时，能够以可信和安全的方式进行。公证项目基于更新框架（The Update Framework）规范开发，更新框架规范定义了一个用于分发和更新内容的安全设计。

Docker 的内容信任框架基本上是一个公证项目与 Docker API 的集成。只需运行寄存服务及公证服务器，机构便能够为用户提供可信任的镜像。然而，公证服务原本的设计是独立运作的，因此它可用于很多不同的场景。

公证服务的一个主要用例是改进常用的 `curl | sh` 命令，通过公证服务可以提升它的安全性和可信度。这个命令一般出现在 Docker 的安装说明中：

```
$ curl -sSL https://get.docker.com/ | sh
```

如果在下载过程中被恶意篡改的话，无论是在服务器上，还是在传输过程中，攻击者能够在受害者的计算机上运行任意命令。使用 HTTPS 能够阻止攻击者在传输过程当中修改数据，但他们仍然可以把下载提前终止，并恶意截断代码。刚才的例子在使用公证服务后将会变成这样：

```
$ curl http://get.docker.com/ | notary verify docker.com/scripts v1 | sh
```

执行 `notary` 命令时，它会将脚本的校验和与公证服务为 docker.com 而设的可信储存中的校验和进行匹配。如果匹配成功，那就能够证明脚本确实来自 docker.com，而且未被篡改。如果匹配失败，公证服务便会退出，那就没有任何数据传给 `sh` 执行了。还有一点值得注意，那就是脚本本身通过不安全的通道传送是没有问题的，而这里使用的正是 HTTP。你不用担心这样做不安全，如果脚本在传输过程中被篡改，那么校验和也会随之变化，公证服务就会报错。

如果你使用未签名的镜像，在下载镜像时，你可以把名字和标签改为摘要，这样做也能够验证镜像的真伪。例如：

```
$ docker pull debian@sha256:f43366bc755696485050ce14e1429c481b6f0ca04505c4a3093d\
fdb4fafb899e
```

撰写这一部分的时候，这个命令会下载 `debian:jessie` 镜像。与使用 `debian:jessie` 标签不一样，这个命令保证下载的镜像永远是同一个（或根本没有对应的镜像）。如果摘要能够以某种方法安全地传送和验证（例如，由可信的一方通过 PGP 签名的电子邮件发送给你），便能够确保镜像的真实性。即使启用了内容信任机制，也可以通过摘要来下载镜像。

如果你需要发布镜像，但又对私人的寄存服务或 Docker Hub 不太信任，那么你在任何时候都可以使用 `docker load` 和 `docker save` 命令来导入和导出镜像。至于镜像分发，你可以在内部网站提供下载镜像的地方，甚至也可以简单使用复制镜像文件的方式。当然，如

果你选择这样做，你可能会发现其实你一步步地把 Docker 的寄存服务和内容信任功能重新实现了一遍。

13.6.3　可复制及可信任的Dockerfile

如果 Dockerfile 每次都能产生完全相同的镜像，这是最理想的，但在现实中却很难做到。一模一样的 Dockerfile 在不同时间产生的镜像都可能不一样。显然这是很有问题的，因为正如前文中所说的，我们将难以确保镜像中包含哪些内容。如果在编写 Dockerfile 时能够遵守以下的规则，那么离构建可复制镜像的目标就不远了。

- 在 `FROM` 指令中必须使用标签。千万不要使用 `FROM redis` 这样的写法，因为它下载的是 `latest` 标签的镜像，而实际的镜像将随着时间而变化，甚至连主版本也会改变。`FROM redis:3.0` 会好一些，但镜像仍然会因为一些小的更新或问题修复而改变（虽然这可能正是你想要的）。如果你想确保每次下载的镜像完全相同，正如之前说过的，唯一的方法就是使用摘要。例如：

  ```
  FROM redis@sha256:3479bbcab384fa343b52743b933661335448f816620368806...
  ```

  使用摘要还可以防止镜像意外损坏或被篡改。
- 使用包管理器安装软件时提供版本号。有时候像 `apt-get install cowsay` 这样的写法是可以接受的，因为 cowsay 不太可能推出更新了，但 `apt-get install cowsay=3.03+dfsg1-6` 这样的写法肯定更好。这同样适用于其他的包管理器，例如 pip，如果可以的话，尽量清楚地列明版本号。如果旧的软件包已被删除，镜像构建就会失败，不过这样就相当于一个警告了。这样做还有一个问题有待解决：软件包有可能还依赖其他的包，而这些依赖关系往往只以 >= 条件来表示，意味着这些包随时有可能改变。要彻底锁定所有软件包的版本，你需要使用如 aptly（http://www.aptly.info/）这类的工具，它可以帮助你制作软件库的快照。
- 验证任何从互联网下载的软件或数据。这意味着你需要使用校验和以及加密签名。这是所有步骤中最重要的一个。如果你没有对下载的数据进行验证的话，那么文件意外损坏或攻击者恶意篡改数据就不容易被察觉。尤其是当软件通过 HTTP 传送时，因为它不能提供防御中间人攻击的保障。接下来将提供这方面的具体建议。

大多数官方镜像的 Dockerfile 在指定版本时，都会使用标签以及对下载的数据进行验证，提供了很好的例子以供参考。通常在指定基础镜像时，它们还会使用特定的标签，但使用包管理器安装软件时则可能不会指定版本号。

在Dockerfile中安全下载软件

大多数情况下，软件供应商都会提供已签名的校验和，以供用户验证下载的数据。举个例子，官方 Node.js 镜像中的 Dockerfile 包含以下内容：

```
RUN gpg --keyserver pool.sks-keyservers.net \
        --recv-keys 7937DFD2AB06298B2293C3187D33FF9D0246406D \
                    114F43EE0176B71C7BC219DD50A3051F888C628D  ❶

ENV NODE_VERSION 0.10.38
ENV NPM_VERSION 2.10.0
```

```
RUN curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/\
node-v$NODE_VERSION-linux-x64.tar.gz" \  ❷
  && curl -SLO "http://nodejs.org/dist/v$NODE_VERSION/SHASUMS256.txt.asc" \  ❸
  && gpg --verify SHASUMS256.txt.asc \
  && grep " node-v$NODE_VERSION-linux-x64.tar.gz\$" SHASUMS256.txt.asc \  ❹
     | sha256sum -c -  ❺
```

❶ 获取用于签署 Node.js 下载文件的 GPG 密钥。这里必须信任这些密钥都是正确的。

❷ 下载 Node.js 的 tar 包。

❸ 下载 tar 包的校验和。

❹ 利用 GPG 验证校验和是否以刚才我们所下载的密钥签署。

❺ 利用 sha256sum 工具检查校验和与压缩包是否匹配。如果 GPG 验证或校验和验证其中一项失败，镜像构建便会中止。

软件包有可能以第三方仓库的方式提供，这时候，把仓库和它的签名密钥添加到系统中，便能保证下载时的安全性。举个例子，官方 Nginx 镜像的 Dockerfile 包含以下内容：

```
RUN apt-key adv --keyserver hkp://pgp.mit.edu:80 \
                --recv-keys 573BFD6B3D8FBC641079A6ABABF5BD827BD9BF62
RUN echo "deb http://nginx.org/packages/mainline/debian/ jessie nginx" \
         >> /etc/apt/sources.list
```

第一行命令获取 Nginx 的签名密钥（并添加到密钥库中），第二行命令把 Nginx 的软件包库添加到系统的软件仓库列表中。之后，Nginx 便可以简单且安全地通过 `apt-get install -y nginx` 命令进行安装（建议指定版本号）。

假如发布者没有提供已签署的软件包或校验和，自行创建一个也很简单。例如，为某个 Redis 版本创建校验和：

```
$ curl -s -o redis.tar.gz http://download.redis.io/releases/redis-3.0.1.tar.gz
$ sha1sum -b redis.tar.gz  ❶
fe1d06599042bfe6a0e738542f302ce9533dde88 *redis.tar.gz
```

❶ 这里创建一个 160 位的 SHA1 校验和。`-b` 参数告诉 `sha1sum` 程序，我们处理的是二进制数据而非文本。

完成对软件的测试和验证后，便可以在 Dockerfile 中加上如下指令：

```
RUN curl -sSL -o redis.tar.gz \
         http://download.redis.io/releases/redis-3.0.1.tar.gz \
    && echo "fe1d06599042bfe6a0e738542f302ce9533dde88 *redis.tar.gz" \
       | sha1sum -c -
```

这个命令把下载文件另存为 redis.tar.gz，并利用 `sha1sum` 来验证校验和。如果验证失败，整个命令也将失败，镜像生成便会中止。

如果发布新版本的频率很高，每次发布都要改动这些内容，这样做的工作量将会是巨大的，因此应该尽量把这个过程自动化。在很多官方镜像的仓库中都能够找到用于自动化更新的 update.sh 脚本，例如这个（https://github.com/docker-library/wordpress/blob/master/update.sh）。

13.7 安全建议

这一节将介绍保护容器部署安全的一些可操作技巧。虽然并非全部建议都适用于所有部署的配置，但你应该熟悉其中你能运用的基本工具。

大部分的建议都是关于如何限制容器的各种方法，使容器无法对主机或其他容器造成不良影响。你必须牢记，主机上的内核资源，包括 CPU、内存、网络、UID 等，都是所有容器间共享的。如果某个容器独占了任何资源，其他需要这个资源的容器就无法正常运行。更糟糕的是，如果一个容器可以利用内核代码中的漏洞，那么它就有可能使主机宕机，或能够入侵主机和其他容器。这可能是不小心造成的后果，也可能是一些有缺陷的程序或恶意攻击行为所导致的，目的是使主机无法正常工作，或把它据为己有。

13.7.1 设置用户

永远不要以 root 用户的身份在容器内运行正式的应用。这句话我必须不断重复：*永远不要以 root 用户的身份在容器内运行正式的应用*。这是因为，如果攻击者攻破了应用程序，那么他就能够获得容器的全部访问权限，包括它的数据和程序。更糟糕的是，如果他能突破容器的限制，他就能够获得主机上的 root 权限。你应该不会在虚拟机和一般的机器上以 root 运行程序，因此在容器内也不要这样做。

为了避免使用 root 用户的身份，你的 Dockerfile 在任何情况下都应该创建一个普通用户，并利用 `USER` 语句或 entrypoint 脚本切换为这个用户。例如：

```
RUN groupadd -r user_grp && useradd -r -g user_grp user
USER user
```

以上的指令创建一个名为 `user_grp` 的用户组，以及一个属于这个组的名为 `user` 的新用户。`USER` 语句对后面的所有指令，以及对利用这个镜像启动的容器生效。因此，如果有些操作需要以 root 执行，例如安装软件，你便需要把 `USER` 指令放在 Dockerfile 靠后的位置，让需要 root 权限的操作首先执行。

很多官方镜像都会创建普通用户，但不一定以 `USER` 指令的方式切换。它们所用的方法可能是在 entrypoint 脚本中以 `gosu` 命令切换用户。例如，官方 Redis 镜像的 entrypoint 脚本是这样的：

```
#!/bin/bash
set -e
if [ "$1" = 'redis-server' ]; then
        chown -R redis .
        exec gosu redis "$@"
fi

exec "$@"
```

脚本中的 `chown -R redis .` 命令将镜像的数据目录下的所有文件改成 redis 用户所有。如果 Dockerfile 中含有 `USER` 指令的话，这个命令将会失败。下一行的 `exec gosu redis "$@"` 以 redis 用户执行 redis 命令。其中的 `exec` 命令把当前的 shell 替换为 redis 进程并成为 PID 1，使它能够接收正确转发过来的任何信号。

使用 gosu，不要使用 sudo

当需要以其他用户身份执行命令的时候，传统上都会使用 sudo。虽然 sudo 功能强大，而且历史悠久，但一些副作用使它在 entrypoint 中应用时不太理想。例如，可以看看在 Ubuntu[12] 容器中执行 sudo ps aux 时会发生什么情况：

```
$ docker run --rm ubuntu:trusty sudo ps aux
USER       PID %CPU ... COMMAND
root         1 0.0      sudo ps aux
root         5 0.0      ps aux
```

我们看到有两个进程，一个属于 sudo，另一个是真正执行的命令。

相反，如果在 Ubuntu 镜像中安装 gosu：

```
$ docker run --rm amouat/ubuntu-with-gosu \
                        gosu root ps aux
USER       PID %CPU ... COMMAND
root         1 0.0      ps aux
```

这里只有一个进程在运行，gosu 执行命令后便完全退出。更重要的是，命令以 PID 1 运行，因此它能够正确地接收发送到容器的任何信号，这一点与 sudo 不同。

如果你的程序必须以 root 身份运行（而你无法修改程序），你应该考虑使用如 sudo、SELinux（参见 13.9.1 节）和 fakeroot 等工具来约束进程。

13.7.2　限制容器联网

容器应仅仅打开在生产环境中必须的端口，并且端口只对所需的容器开放。虽然听起来很简单，但实际操作并非如此，因为在默认情况下，无论端口是否已被明确声明开放，容器之间都可以互相通信。通过 Netcat 工具[13] 我们可以很容易的证实这一点：

```
$ docker run --name nc-test -d amouat/network-utils nc -l 5001  ❶
f57269e2805cf3305e41303eafefaba9bf8d996d87353b10d0ca577acc731186
$ docker run \
   -e IP=$(docker inspect -f {{.NetworkSettings.IPAddress}} nc-test) \
   amouat/network-utils sh -c 'echo -n "hello" | nc -v $IP 5001'  ❷
Connection to 172.17.0.3 5001 port [tcp/*] succeeded!
$ docker logs nc-test
hello
```

❶ 告诉 Netcat 监听 5001 端口，并打印所有接收到的信息。

❷ 利用 Netcat 发送“hello”到第一个容器。

尽管没有端口被发布或开放，第二个容器还是能够连接到 nc-test。运行 Docker 守护进程时加上 --icc=false 参数可以改变这个行为。这个参数的作用是关闭容器间的通信，可以防止被入侵的容器继而攻击其他容器。选项关闭后，任何已明确连接的容器仍然能够进行通信。

注 12：这里使用的是 Ubuntu 而不是 Debian，因为 Ubuntu 的镜像已默认包含 sudo。

注 13：这里使用的是 OpenBSD 版本。

Docker 通过设置 IPtables 规则来控制容器间的通信（要求启动守护进程时设置 `--iptables` 参数，这应该是默认行为）。

下面的例子演示了守护进程设置 `--icc=false` 的效果：

```
$ cat /etc/default/docker | grep DOCKER_OPTS=
DOCKER_OPTS="--iptables=true --icc=false"  ❶
$ docker run --name nc-test -d --expose 5001 amouat/network-utils nc -l 5001
d7c267672c158e77563da31c1ee5948f138985b1f451cd2222cf248006491139
$ docker run \
   -e IP=$(docker inspect -f {{.NetworkSettings.IPAddress}} nc-test)
   amouat/network-utils sh -c 'echo -n "hello" | nc -w 2 -v $IP 5001'  ❷
nc: connect to 172.17.0.10 port 5001 (tcp) timed out: Operation now in progress
$ docker run \
   --link nc-test:nc-test \
   amouat/network-utils sh -c 'echo -n "hello" | nc -w 2 -v nc-test 5001'
Connection to nc-test 5001 port [tcp/*] succeeded!
$ docker logs nc-test
hello
```

❶ 在 Ubuntu 上，Docker 守护进程通过 /etc/default/docker 中的 `DOCKER_OPTS` 参数进行配置。
❷ `-w 2` 参数告诉 Netcat 超时时间为两秒。

第一个命令连接失败，因为容器间的通信已关闭，而且没有用到 Docker 连接。第二个命令连接成功，因为我们添加了 Docker 连接。如果想了解背后的工作原理，可以分别在 Docker 连接存在与不存在的情况下，尝试在主机上运行 `sudo iptables -L -n`。

在主机上发布端口时，Docker 默认在所有接口（0.0.0.0）上发布。不过，你可以明确指定希望绑定的接口：

```
$ docker run -p 87.245.78.43:8080:8080 -d myimage
```

这样做限制通信只能在指定的接口上进行，从而缩小了攻击面。

13.7.3　删除setuid和setgid的二进制文件

你的应用程序很可能不需要使用任何 setuid 或 setgid 的二进制文件[14]。如果可以禁用或删除这些二进制文件，那么就能阻止攻击者利用它们来发动提权攻击。

要找出镜像中所有的这些二进制文件，可以执行 `find / -perm +6000 -type f -exec ls -ld {} \;`。例如：

```
$ docker run debian find / -perm +6000 -type f -exec ls -ld {} \; 2> /dev/null
-rwsr-xr-x 1 root root 10248 Apr 15 00:02 /usr/lib/pt_chown
-rwxr-sr-x 1 root shadow 62272 Nov 20 2014 /usr/bin/chage
-rwsr-xr-x 1 root root 75376 Nov 20 2014 /usr/bin/gpasswd
-rwsr-xr-x 1 root root 53616 Nov 20 2014 /usr/bin/chfn
-rwsr-xr-x 1 root root 54192 Nov 20 2014 /usr/bin/passwd
-rwsr-xr-x 1 root root 44464 Nov 20 2014 /usr/bin/chsh
```

注 14：setuid 和 setgid 的二进制文件以文件的所有者身份运行，而并非当前用户。它们通常用于允许用户执行某些需要临时提升权限的任务，例如设置密码。

```
-rwsr-xr-x 1 root root 39912 Nov 20 2014 /usr/bin/newgrp
-rwxr-sr-x 1 root tty 27232 Mar 29 22:34 /usr/bin/wall
-rwxr-sr-x 1 root shadow 22744 Nov 20 2014 /usr/bin/expiry
-rwxr-sr-x 1 root shadow 35408 Aug 9 2014 /sbin/unix_chkpwd
-rwsr-xr-x 1 root root 40000 Mar 29 22:34 /bin/mount
-rwsr-xr-x 1 root root 40168 Nov 20 2014 /bin/su
-rwsr-xr-x 1 root root 70576 Oct 28 2014 /bin/ping
-rwsr-xr-x 1 root root 27416 Mar 29 22:34 /bin/umount
-rwsr-xr-x 1 root root 61392 Oct 28 2014 /bin/ping6
```

可以执行 `chmod a-s` 命令来移除二进制文件的 suid 设置，从而消除它们的危害。例如，可以通过下面的 Dockerfile 创建一个已清除 setuid/setgid 的 Debian 镜像：

```
FROM debian:wheezy

RUN find / -perm +6000 -type f -exec chmod a-s {} \; || true  ❶
```

❶ 命令行中的 `|| true` 语句用于忽略 `find` 命令的任何错误。

现在构建镜像并运行它：

```
$ docker build -t defanged-debian .
...
Successfully built 526744cf1bc1
docker run --rm defanged-debian \
  find / -perm +6000 -type f -exec ls -ld {} \; 2> /dev/null | wc -l
0
$
```

可能你的 Dockerfile 比你的应用程序更依赖 setuid/setgid 二进制文件。因此，你可以等到 Dockerfile 接近结尾的部分才执行这一步，好让 setuid/setgid 二进制文件能够首先执行，但必须在改变用户之前（如果应用以 root 权限运行，那么删除 setuid 二进制文件是毫无意义的）。

13.7.4　限制内存使用

限制内存使用，可以防止 DoS 攻击和应用程序的内存泄漏。内存泄漏将使主机的内存慢慢消耗殆尽（如果遇到有这种问题的应用程序，你可以让它自动重启以维持服务水平）。

`docker run` 的 `-m` 和 `--memory-swap` 参数可以用来限制容器能够使用的内存和虚拟内存容量。但这个参数的名字不太好理解，其实 `--memory-swap` 设置的是*总共*的内存容量，即物理内存*加上*虚拟内存，而不仅仅是虚拟内存。默认设置是没有限制内存使用的。如果只用了 `-m` 参数而没有 `--memory-swap`，那么 `--memory-swap` 就会被设置为 `-m` 参数的一倍。举个例子说明会比较容易理解。下面我们将使用 `amouat/stress` 镜像，因为它包含了 Unix 的 stress 工具（http://people.seas.harvard.edu/~apw/stress/），可以用来测试进程占用资源的情况。在这个例子中，我们会告诉它分配一定的内存：

```
$ docker run -m 128m --memory-swap 128m amouat/stress \
     stress --vm 1 --vm-bytes 127m -t 5s  ❶
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: info: [1] successful run completed in 5s
```

```
$ docker run -m 128m --memory-swap 128m amouat/stress \
    stress --vm 1 --vm-bytes 130m -t 5s ❷
stress: FAIL: [1] (416) <-- worker 6 got signal 9
stress: WARN: [1] (418) now reaping child worker processes
stress: FAIL: [1] (422) kill error: No such process
stress: FAIL: [1] (452) failed run completed in 0s
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
$ docker run -m 128m amouat/stress \
    stress --vm 1 --vm-bytes 255m -t 5s ❸
stress: info: [1] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: info: [1] successful run completed in 5s
```

❶ 这些参数告诉 stress 运行一个要求分配 127MB 内存的进程，超时时间为 5 秒。
❷ 这一次尝试请求 130MB 内存，因为容器只有 128MB 的内存，所以内存请求失败。
❸ 这一次尝试请求 255MB 内存，因为 `--swap-memory` 的值默认为 256MB，所以内存请求成功。

13.7.5 限制CPU使用

如果攻击者可以取得一个或一组容器，并把主机上的 CPU 完全占用的话，攻击者便能够使主机上的其他容器无法分配适当的 CPU 资源，从而导致系统被 DoS 攻击瘫痪。

Docker 控制 CPU 占有率是依靠*相对*的比重，默认值为 1024。因此，所有容器默认都有相同的 CPU 使用份额。

举个例子就容易理解了。我们将利用之前使用过的 `amouat/stress` 镜像来启动 4 个容器，但这一次它们将尝试占用尽可能多的 CPU 资源，而不是内存。

```
$ docker run -d --name load1 -c 2048 amouat/stress
912a37982de1d8d3c4d38ed495b3c24a7910f9613a55a42667d6d28e1da71fe5
$ docker run -d --name load2 amouat/stress
df69312a0c959041948857fca27b56539566fb5c7cda33139326f16485948bc8
$ docker run -d --name load3 -c 512 amouat/stress
c2675318fefafa3e9bfc891fa303a16e72caf221ec23a4c222c2b889ea82d6e2
$ docker run -d --name load4 -c 512 amouat/stress
5c6e199423b59ae481d41268c867c705f25a5375d627ab7b59c5fbfbcfc1d0e0
$ docker stats $(docker inspect -f {{.Name}} $(docker ps -q))
CONTAINER           CPU % ...
/load1              392.13%
/load2              200.56%
/load3              97.75%
/load4              99.36%
```

在这个例子中，容器 `load1` 的比重是 2048，`load2` 的比重是默认的 1024，而另外两个容器的比重为 512。我的计算机是 8 核的，因此一共有 800%的 CPU 可以分配，最终 `load1` 得到大约一半的 CPU，`load2` 得到四分之一，`load3` 和 `load4` 各获得八分之一。如果运行的容器只有一个，那么它就能够用尽所有资源。

相对比重的设计意味着在默认设置的情况下，没有任何容器可以占用过多资源。然而，你可能有很多容器“组”，它们总共被分配的 CPU 比其他容器要多，在这种情况下，你可以

指定容器组中的容器使用较低的默认值，以保证资源能够被公平使用。如果你确实有必要指定 CPU 份额，务必把默认值考虑进去，不要使你的容器获得过多的 CPU，而应该让没有明确设置配额的容器都能获得合理份额的资源。

另外，CPU 可以使用完全公平调度器（Completely Fair Scheduler，CFS）进行分配，方法是利用 `--cpu-period` 和 `--cpu-quota` 参数。在这个方法中，容器会被分配一个在一段既定时间内能够使用的 CPU 配额（以微秒为单位）。如果容器在既定时间内使用的 CPU 超过了配额，它就必须等到下一个时间段才能够继续执行。例如：

```
$ docker run -d --cpu-period=50000 --cpu-quota=25000 myimage
```

假设系统只有一颗 CPU，这个容器将允许每 50 毫秒使用一半的 CPU。有关 CFS 的更多信息，请参阅 Linux 内核文件（https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt）。

13.7.6 限制重新启动

如果容器不断崩溃之后又不断重新启动，那将会造成大量系统时间和资源浪费，更有可能导致系统无法访问。为了阻止这一状况发生，只需把重启策略由 `always` 改为 `on-failure` 即可。例如：

```
$ docker run -d --restart=on-failure:10 my-flaky-image
...
```

在这个例子中，Docker 只会最多重启容器 10 次。当前的重启次数可以通过 `docker inspect` 的 `.RestartCount` 元数据找到：

```
$ docker inspect -f "{{ .RestartCount }}" $(docker ps -lq)
0
```

Docker 重新启动容器的间隔时间采用指数递增的方式（第一次重启等待 100 毫秒，下一次是 200 毫秒，再下一次是 400 毫秒，以此类推）。对于尝试利用重启功能引发的 DoS 攻击，这是个有效的防止措施。

13.7.7 限制文件系统

阻止攻击者写入文件系统，可以抵御对系统的一些攻击，至少大体上使攻击者难以得逞。他们将无法把脚本文件写进去，从而诱骗你的程序执行它，或者重写敏感数据或配置文件。

从 Docker 1.5 开始，可以在执行 `docker run` 时使用 `--read-only` 参数，把容器的文件系统设置为完全只读：

```
$ docker run --read-only debian touch x
touch: cannot touch 'x': Read-only file system
```

你也可以在数据卷参数之后加上 `:ro` 来达到类似效果：

```
$ docker run -v $(pwd):/pwd:ro debian touch /pwd/x
touch: cannot touch '/pwd/x': Read-only file system
```

大部分应用都需要写入文件，因此它们在完全只读的环境中无法运作。如果遇到这种情况，可以找出应用程序需要写入的文件夹和文件，然后只针对这些文件以数据卷的方式加载它们。

采用这种方法能使系统审核从中受益良多。如果可以肯定容器的的文件系统与产生它的镜像完全相同，那么在执行系统审核时，只需要对镜像进行离线审核即可，就不用对每个容器逐一检查了。

13.7.8 限制内核能力

Linux 内核定义了一些特殊的权限，即所谓的能力（capability），把这些权限分配给进程后，进程就能够获得更多的系统权限。可用的内核能力涵盖广泛的功能，从更改系统时间到开启网络套接字。从前，一个进程要么拥有完全的 root 权限，要么只是一个普通用户，没有介于两者之间的情况。对于 ping 这样的程序而言，这种设计是尤其不便的，它必须拥有 root 的权限，只因需要打开一个原始网络套接字（raw network socket）。这意味着，如果 ping 有一个小 bug，它就可能被攻击者利用，并取得系统所有的 root 权限。随着内核能力的出现，我们就可以开发出一个只具有原始网络套接字能力的 ping，而无需完整的 root 权限了。这就意味着潜在的攻击者能够从程序的漏洞中得到的东西也会少很多。

默认情况下，Docker 运行容器时只授权部分内核能力。[15] 例如，容器通常不允许使用如 GPU 或声卡的硬件设备，或加载内核模块。如果需要扩展容器的权限，可以在执行 `docker run` 时加上 `--privileged` 参数。

就安全方面而言，我们真正想做到的是尽可能限制容器的能力。我们可以通过 `--cap-add` 和 `--cap-drop` 参数来控制容器可用的功能。举个例子，如果我们希望能够更改系统时间（不要真的照着做，除非你不怕把系统搞乱）：

```
$ docker run debian date -s "10 FEB 1981 10:00:00"
Tue Feb 10 10:00:00 UTC 1981
date: cannot set date: Operation not permitted
$ docker run --cap-add SYS_TIME debian date -s "10 FEB 1981 10:00:00"
Tue Feb 10 10:00:00 UTC 1981
$ date
Tue Feb 10 10:00:03 GMT 1981
```

从这个例子可以看到，给容器加上 `SYS_TIME` 权限前，系统日期是无法修改的。由于系统时间是一个没有命名空间特性的内核功能，在容器内设定时间将同时改变主机和其他容器的时间。[16]

更严格的做法是先把所有权限清空，然后只把我们需要的加回去：

注 15：它们包括 `CHOWN`、`DAC_OVERRIDE`、`FSETID`、`FOWNER`、`MKNOD`、`NET_RAW`、`SETGID`、`SETUID`、`SETFCAP`、`SETPCAP`、`NET_BIND_SERVICE`、`SYS_CHROOT`、`KILL` 和 `AUDIT_WRITE`。被放弃的内核能力中比较突出的包括（但不限于）`SYS_TIME`、`NET_ADMIN`、`SYS_MODULE`、`SYS_NICE` 和 `SYS_ADMIN`。有关内核能力的完整信息，请参阅 `man capabilities`。

注 16：如果你跟着例子做的话，那么你的系统将无法正常运作，直到你把系统时间调整正确。你可以尝试执行 `sudo ntpdate` 或 `sudo ntpdate-debian` 改回正确的时间。

```
$ docker run --cap-drop all debian chown 100 /tmp
chown: changing ownership of '/tmp': Operation not permitted
$ docker run --cap-drop all --cap-add CHOWN debian chown 100 /tmp
```

这样做将大幅度提高系统的安全性。即使攻击者能成功入侵容器，他能调用的内核函数也很大程度上受到限制。但是，目前还存在一些问题。

- 你怎么知道哪些权限是可以放弃的？不断尝试与修正似乎是最简单的方法，但你有可能放弃的是一个你的程序很少才用到的权限。如果你的容器有完整的测试套件，或者你采用的是微服务架构，其中涉及的代码和部件都比较少，那么确定所需的权限就比较容易。
- 你可能会认为，目前对于内核能力的分类还不够理想或粒度不够精细。尤其是 `SYS_ADMIN` 这个能力包含了非常多的功能，内核开发者似乎在找不到（或嫌麻烦不想找）更合适的能力时便把它当作默认值来用。实际上，这样做违背了内核能力设计的初衷，那就是为了消除管理用户与普通用户这种二元对立。

13.7.9　应用资源限制

Linux 内核定义了一些用于进程的资源限制（ulimits），例如限制进程允许 fork 生成的子进程数量，或允许开启的文件描述符数量。这些限制也可以用于 Docker 容器，方法是在执行 `docker run` 时加上 `--ulimit` 选项，或在启动 Docker 守护进程时加上 `--default-ulimit` 选项给所有容器设置默认值。选项的参数中包括两个数值，一个是非强制限制（soft limit），另一个是强制限制（hard limit），中间以冒号分隔，它们的确切作用取决于参数中指定要限制哪种资源。如果只提供一个数值，它就会被同时用作非强制和强制限制的值。

关于可用于参数的限制及它们的意义，可以在 `man setrlimit` 中找到完整的介绍（注意 `as` 限制不适用于容器）。下面是一些特别值得注意的限制。

cpu

把 CPU 时间限制为给定的参数，以秒为单位。非强制限制达到后，会送出一个 `SIGXCPU` 信号给容器；强制限制达到后，就会送出 `SIGKILL` 信号。这里以之前在 13.7.4 节和 13.7.5 节中使用过的 stress 工具作为例子，看看如何最大限度地提高 CPU 的使用率：

```
$ time docker run --ulimit cpu=12:14 amouat/stress stress --cpu 1
stress: FAIL: [1] (416) <-- worker 5 got signal 24
stress: WARN: [1] (418) now reaping child worker processes
stress: FAIL: [1] (422) kill error: No such process
stress: FAIL: [1] (452) failed run completed in 12s
stress: info: [1] dispatching hogs: 1 cpu, 0 io, 0 vm, 0 hdd

real  0m12.765s
user  0m0.247s
sys 0m0.014s
```

通过设置 `ulimit`，容器使用了 12 秒 CPU 后便被终止。

这个方法适用于限制那些由其他进程启动的容器（例如代替用户执行运算工作），限定它们能够使用多少 CPU 时间。以这种方式限制 CPU 的使用，可以有效缓解 DoS 攻击。

nofile

指定容器中能够同时开启最多的文件描述符[17]数量。同样，这个限制可以用来抵御 DoS 攻击，并确保攻击者无法对容器或数据卷进行读取或写入（请注意，如果要做到这一点，你需要将 nofile 的数值设置为最大数目加 1）。举例如下：

```
$ docker run --ulimit nofile=5 debian cat /etc/hostname
b874469fe42b
$ docker run --ulimit nofile=4 debian cat /etc/hostname
Timestamp: 2015-05-29 17:02:46.956279781 +0000 UTC
Code: System error

Message: Failed to open /dev/null - open /mnt/sda1/var/lib/docker/aufs...
```

可以看到，虽然 cat 只需开启一个文件描述符，但操作系统则会用到好几个。确定你的程序实际需要使用多少文件描述符是很困难的，你可以在设置参数时指定一个比较大的数目，这样做总比默认的 1 048 576 个文件描述符上限能够提供多一点的 DoS 保护。

nproc

指定运行容器的用户能够创建进程的最大数目。表面上这个参数很有用，因为它可以防止 fork 炸弹和其他类型的攻击。可惜的是，nproc 的限制不是针对每个容器，而是针对运行这个容器的用户的所有进程而设定的。例如：

```
$ docker run --user 500 --ulimit nproc=2 -d debian sleep 100
92b162b1bb91af8413104792607b47507071c52a2e3128f0c6c7659bfbb84511
$ docker run --user 500 --ulimit nproc=2 -d debian sleep 100
158f98af66c8eb53702e985c8c6e95bf9925401c3901c082a11889182bc843cb
$ docker run --user 500 --ulimit nproc=2 -d debian sleep 100
6444e3b5f97803c02b62eae601fbb1dd5f1349031e0251613b9ff80871555664
FATA[0000] Error response from daemon: Cannot start container 6444e3b5f9780...
[8] System error: resource temporarily unavailable
$ docker run --user 500 -d debian sleep 100
f740ab7e0516f931f09b634c64e95b97d64dae5c883b0a349358c5995806e503
```

因为已经有两个属于 UID 500 用户的进程，所以第三个容器便无法启动。然而，只需要把 --ulimit 选项拿掉，这个用户就能够继续创建进程了。虽然这是一个重大缺陷，但对于同一用户只会使用少量容器的情况下，nproc 限制还是有点用的。

另外，请注意 nproc 限制不适用于 root 用户。

13.8 运行加固内核

除了一般的安全措施，例如保持主机的操作系统为最新，以及保证问题补丁都已安装，还可以考虑采用 grsecurity（https://grsecurity.net/）和 PaX（https://pax.grsecurity.net/）提供的补丁，运行一个加固内核（hardened kernel）。针对攻击者通过修改内存来操纵程序执行（例如缓冲区溢出攻击）这种问题，PaX 对系统提供了额外保护。它的实现方式是将内

注 17：文件描述符是一个指向系统上用于记录已开启文件列表的指针。每当需要对一个文件进行访问时，该列表便会创建一个新条目，其中会记录文件的访问模式（读、写等）以及指向该文件的指针。

存中的程序代码标记为不可写入，并将数据标记为不可执行。此外，内存使用的安排被随机化，使得试图把代码重定向至现有函数（例如公共库中的系统调用）的恶意攻击得以缓解。grsecurity 的设计是与 PaX 互补，它的补丁包括基于角色的访问控制（RBAC）、审核以及各种琐碎功能。

如果你希望使用 PaX 和 grsecurity，你可能需要亲自为内核打补丁和进行编译。你可能会以为很复杂，其实不然，而且你可以找到大量的参考资源 [参见维基教科书（https://en.wikibooks.org/wiki/Grsecurity/Configuring_and_Installing_grsecurity）和 InsanityBit（http://www.insanitybit.com/2012/05/31/compile-and-patch-your-own-secure-linux-kernel-with-pax-and-grsecurity/）]。

这些安全增强功能可能会导致某些应用程序无法工作。PaX 会与任何运行时生成代码的程序产生冲突。此外，额外的安全检查和措施将产生少量系统开销。最后，如果你使用的是已经预先编译好的内核，你必须确保它的版本足够新，这样才能支持你的 Docker 版本。

13.9 Linux安全模块

Linux 内核定义了 Linux 安全模块（Linux Security Module，LSM）接口，为了实施特定的安全策略而开发的各种内核模块都需要利用这个接口。本书写作之际已经出现了数个 Linux 安全模块，其中包括 AppArmor、SELinux、Smack 以及 TOMOYO Linux。它们都超越了传统文件级别的访问控制，对于进程和用户的访问权限，它们提供了另一重的安全检查。

通常与 Docker 一起使用的模块有 SELinux（一般用于红帽和它的衍生版）和 AppArmor（一般用于 Ubuntu 和 Debian 发行版）。接下来就看看这两个内核模块。

13.9.1 SELinux

SELinux，全称为*安全增强* Linux（Security Enhanced Linux），这个模块是由美国国家安全局（National Security Agency，NSA）开发，用以实现其宣称的强制访问控制（Mandatory Access Control，MAC），而在标准的 Unix 上使用的是自主访问控制（Discretionary Access Control，DAC）。以比较浅显的语言表述的话，由 SELinux 实施的访问控制和标准 Linux 的访问控制有两个主要区别。

- SELinux 是基于*类型*（type）来执行它的访问控制，类型基本上是赋予进程和对象（文件、套接字等）的标签。如果 SELinux 的策略是禁止类型 A 的进程访问类型 B 的对象，那么无论原来对象的文件权限或用户的访问权限是否允许，这个访问动作都将被禁止。SELinux 的权限检查发生在正常文件权限检查之后。
- 可以实施多重密级，类似政府中制定的机密（confidential）、秘密（secret）以至绝密（top-secret）级别。较低级别的进程不能读取由较高级别的进程写入的文件，无论文件在文件系统中处于什么位置，或文件权限是什么。因此，绝密级别的进程可以在 `/tmp` 目录中写入一个 `chmod 777` 权限的文件，但机密级别的进程仍然无法访问这个文件。在 SELinux 中，这个机制被称为*多级别安全制度*（Multi-level security，MLS），另外还有一个密切相关的概念称为*多类别安全制度*（Multi-category security，MCS）。在 MCS 的

设计中，进程和对象被赋予各种类别，类别不匹配的将被拒绝资源访问。与 MLS 不同，类别之间没有重叠，并且不按等级划分。MCS 可以针对类型中的一个子集，限制它们的资源访问权限（例如，通过使用一个独立的类别，我们可以限制某个资源只供单一进程使用）。

基于红帽的发行版默认自带 SELinux，在其他发行版上安装也很简单。执行 sestatus 命令能够检查 SELinux 是否正在运行。如果该命令存在，它会告诉你 SELinux 的状态是启用还是禁用，处于许可（permissive）模式还是强制执行（enforcing）模式。当 SELinux 处于许可模式时，它只会把不正当的访问记录下来，但不会强制执行它。

Docker 的默认 SELinux 安全策略旨在保护主机不受到来自容器的攻击，以及保护容器不受到来自其他容器的攻击。容器被赋予的默认进程类型为 svirt_lxc_net_t，而容器允许访问的文件被赋予 svirt_sandbox_file_t 类型。这个安全策略强制规定容器只能读取和执行主机上 /usr 的文件，并且不能在主机上写入任何文件。每个容器还被分配一个唯一的 MCS 类别编号，这是为了防止容器被突破后可以读取由其他容器写入的文件或资源。

> **启用 SELinux**
>
> 如果你正在使用一个基于红帽的发行版，那么 SELinux 应已安装。在命令行执行 sestatus 命令便可以检查 SELinux 是否已经启用，以及是否处于强制执行模式。编辑 /etc/selinux/config 配置文件并设定 SELINUX=enforcing，便能启用 SELINUX 并将其设置为强制模式。
>
> 你还需要确定 Docker 守护进程已启用 SELinux 的支持，检查它在执行时有没有使用 --selinux-enabled 参数。如果没有，你需要把它添加到 /etc/sysconfig/docker 文件中。
>
> 启用 SELinux 还必须使用 devicemapper 存储驱动程序。本书写作之际，SELinux 暂时还未兼容 Overlay 和 BTRFS，这个功能仍在开发中。
>
> 关于如何在其他发行版上安装 SELinux，请参阅相关文档。有一点你必须注意，SELinux 需要给文件系统上的所有文件赋予标签，这需要一定时间。千万不要一时心血来潮装上 SELinux！

启用 SELinux 对使用数据卷的容器有直接和重大的影响。如果你安装了 SELinux，那么将默认不能读取或写入数据卷：

```
$ sestatus | grep mode
Current mode:                   enforcing
$ mkdir data
$ echo "hello" > data/file
$ docker run -v $(pwd)/data:/data debian cat /data/file
cat: /data/file: Permission denied
```

通过检查文件夹的安全上下文就能知道原因：

```
$ ls --scontext data
unconfined_u:object_r:user_home_t:s0 file
```

原因是数据的标签与容器的标签不匹配。解决方法是利用 chcon 命令把容器的标签应用在

数据上，实际产生的效果等同于通知系统，我们期待容器将读取这些文件：

```
$ chcon -Rt svirt_sandbox_file_t data
$ docker run -v $(pwd)/data:/data debian cat /data/file
hello
$ docker run -v $(pwd)/data:/data debian sh -c 'echo "bye" >> /data/file'
$ cat data/file
hello
bye
$ ls --scontext data
unconfined_u:object_r:svirt_sandbox_file_t:s0 file
```

需要注意的是，如果你只针对文件执行 `chcon` 而不是针对父目录，这将导致你只能读取文件，但无法写入。

从 1.7 版开始，如果在加载数据卷时加上 `:Z` 或 `:z` 后缀，那么 Docker 便会自动给数据卷重新赋予标签，让容器能正常使用数据卷。`:z` 使所有容器都能够使用被重新赋予标签的数据卷（这是数据容器所必需的，因为它需要与多个容器共享数据卷），而使用 `:Z` 的数据卷只能被该容器使用。例如：

```
$ mkdir new_data
$ echo "hello" > new_data/file
$ docker run -v $(pwd)/new_data:/new_data debian cat /new_data/file
cat: /new_data/file: Permission denied
$ docker run -v $(pwd)/new_data:/new_data:Z debian cat /new_data/file
hello
```

你也可以使用 `--security-opt` 选项更改容器的标签，或禁止给容器赋予标签：

```
$ touch newfile
$ docker run -v $(pwd)/newfile:/file --security-opt label:disable \
       debian sh -c 'echo "hello" > /file'
$ cat newfile
hello
```

一个有趣的 SELinux 标签的用法是把一个特定标签赋予容器，使容器强制执行某个特定的安全策略。例如，你可以为 Nginx 容器制定一个只允许它在 80 端口和 443 端口进行通信的安全策略。

注意，在容器内将无法运行 SELinux 的命令。SELinux 在容器内将被视为关闭，以防止应用程序和用户试图执行设置 SELinux 策略这样的命令，而主机也会禁止这些命令运行。

你应该能够找到很多工具和文章帮助你制定 SELinux 的安全策略。其中有一个值得注意的工具叫作 `audit2allow`，它可以把许可模式下运行程序时产生的日志转变为可用于强制执行模式的策略，而不会影响程序的执行。

SELinux 的未来看上去充满希望。随着越来越多的选项和默认的实现被加入 Docker，运行 SELinux 保护的部署应该会变得更简单。最好只需一个简单的参数，就能够利用 MCS 创建适合用于处理敏感信息的秘密和绝密容器。遗憾的是，目前 SELinux 的用户体验并不很好。一般初次接触 SELinux 的用户总是看到“Permission Denied”（因未授权拒绝访问）错误，而且对哪里出错毫无头绪，对如何解决也一筹莫展。开发者不愿意在使用 Docker 时开启 SELinux，导致开发环境与生产环境不一致，但这正是 Docker 致力于解决的问题。

如果你希望或需要 SELinux 提供额外保护，你只能暂时忍耐目前的状况，直到情况有所改善。

13.9.2 AppArmor

AppArmor 的优点在于它比 SELinux 简单得多，但这同时也是它的缺点。基本上你不用做任何事情，它就能正常运作，并且不会对你的程序有任何干扰，但与 SELinux 相比，AppArmor 无法提供相同粒度的保护。AppArmor 的工作方式是将描述安全策略的配置文件应用到进程上，限制进程能够使用哪些 Linux 内核能力和拥有哪些文件访问权限。

假如你的主机运行的是 Ubuntu，AppArmor 很可能已经在运作中。可以执行 `sudo apparmor_status` 证实一下。Docker 将自动把 AppArmor 的安全策略应用在每个被启动的容器上。默认的安全策略能够阻挡流氓容器试图访问各种系统资源，配置文件通常位于 /etc/apparmor.d/docker。本书写作之际，修改默认的配置文件是没有用的，因为 Docker 守护程序在重启后会把它重写。

如果 AppArmor 干扰容器的正常运作，可以把那个容器的 AppArmor 关掉，方法是在执行 `docker run` 时使用 `--security-opt="apparmor:unconfined"` 参数。如果希望容器使用另一个安全配置，可以在执行 `docker run` 时使用 `--security-opt="apparmor:PROFILE"` 参数，而其中的 `PROFILE` 是 AppArmor 已加载的安全配置文件名称。

13.10 审核

定期对你的容器和镜像进行审核或审查，是保证你的系统处于干净和最新的状态的一个很好的方法，而且通过重复检验，可以确保没有尚未察觉的安全事故。审核一个容器化系统，应检查所有正在运行的容器，确保使用的镜像是最新的，而这些镜像使用的软件也应该是最新的和没有安全问题的。如果发现容器与创建该容器的镜像之间有任何差异，那就应该识别问题并进行详细检查。此外，审核还应该包括那些并非容器化系统独有的地方，例如检查访问日志、文件权限和数据完整性。如果大部分审核工作能够实现自动化，它们就可以定期执行，问题也能够尽早发现。

检查容器的时候，不一定需要登入每个容器并对它们逐一进行检查，我们可以审核用于构建容器的镜像，通过 `docker diff` 检查容器是否与镜像存在任何差异。如果使用的是只读文件系统就更好了（参见 13.7.7 节），这样可以确保容器内没有任何东西被更改过。

就最低限度而言，检查时应确认你使用的软件版本是最新的，并且确保最新的安全补丁已经用上。你应该检查每个镜像，并且如果通过 `docker diff` 命令发现任何被修改过的文件，这些文件也应该被检查。如果使用了数据卷，你还需要审核每个数据卷目录。

如果你能够运行一个极简的镜像，其中只包含应用程序必需的文件和程序库，那么审核的工作量将会大幅度减少。

正如你会对一台普通的主机或虚拟机所做的那样，主机系统也需要进行审核。主机的内核是所有容器共用的，因此在容器化系统中，确保内核已正确安装安全补丁就变得尤其重要。

目前已经有一些工具可以用于审核容器系统，我期望接下来的几个月将会有更多工具面

世。值得注意的是，Docker 发布了 Docker Bench for Security（https://github.com/docker/docker-bench-security），这个工具可以检查系统是否遵循 Docker Benchmark 文件中的建议，该文件由 Docker 与互联网安全中心（Center for Internet Security，CIS，https://benchmarks.cisecurity.org/）联合制定，文件中提出了很多安全方面的建议。此外，开源审核工具 Lynis（https://cisofy.com/lynis/）也有一些针对运行 Docker 的检查。

13.11 事件响应

一旦发生任何状况，你可以利用 Docker 的一些功能对事件快速作出反应，并查出问题的根源所在。其中一个重要的命令是 `docker commit`，它能够快速获取遭受攻击的系统的快照，而 `docker diff` 和 `docker logs` 则可以揭示攻击者曾经作出的更改。

当处理遭受攻击的系统时，一个必须回答的问题是："容器突破有没有发生？"（换句话说，攻击者获得了主机的访问权限了吗？）。如果你认为有这个可能性，甚至应该已经发生了，那么主机上的所有东西都必须完全删除，所有容器只能从镜像重新创建（在某种补救措施真正实施到位之前）。如果你能够确定攻击完全被容器隔离，只需停止容器并把它更换即可。（绝对不能把已被成功入侵的容器重新投入服务，即使容器中有一些基础镜像没有的数据或修改也不行，因为这个容器已经无法信任了。）

对容器实施限制措施可能是防御攻击的有效手段，譬如放弃某些内核能力，或把文件系统设为只读。

一旦紧急情况已被处理，并且应对攻击的措施已实施到位，便可以对问题镜像进行分析，以确定攻击的根本原因和影响程度。

有关如何制定事件响应的有效安全策略，请参阅由 CERT 发布的"Steps for Recovering from a UNIX or NT System Compromise"（https://www.cert.org/historical/tech_tips/win-UNIX-system_compromise.cfm），以及 ServerFault 网站上的建议（https://serverfault.com/questions/218005/how-do-i-deal-with-a-compromised-server）。

13.12 未来特性

与安全相关的几个 Docker 特性正在开发中。由于 Docker 已经提升了这些功能的优先级别，当你读到这里的时候，或许已经能够用上它们了。

seccomp

Linux 的 seccomp（或称为 secure computing mode）机制可用于限制进程能够调用的内核系统函数。seccomp 最为人熟知的应用领域是网页浏览器，包括 Chrome 和 Firefox，它们以沙盒的方式来隔离插件。Docker 与 seccomp 整合后，可以使容器只能调用某部分系统函数。在 Docker 与 seccomp 的整合提议中，32 位的函数、过去的网络函数，以及容器一般不需要的系统函数将被默认拒绝。此外，在系统运行时，你可以明确声明需要拒绝或允许的函数。例如，下面的命令将允许容器调用 `clock_adjtime`，因为网络时间协议（Network Time Protocol，NTP）的守护进程需要它来进行系统时间同步：

```
$ docker run -d --security-opt seccomp:allow:clock_adjtime ntpd
```

用户命名空间

> 如前文所述，已经有多个提案涉及如何改善用户命名空间的问题，尤其是 root 用户。可以预期在不久的将来，Docker 将支持把 root 用户映射到主机上的普通用户。

此外，我希望 Docker 能够把各种不同的安全工具进行整合，例如以安全配置文件的方式应用在容器上。目前，安全工具和选项之间还有大量的功能重叠（例如，为了限制文件存取，可以使用 SELinux、禁止相关的内核能力，以及使用 `--read-only` 参数）。

13.13　总结

正如我们在本章中所看到的，确保系统安全要考虑很多方面。首要的安全策略是遵循纵深防御和最低权限的原则。这能够确保，即使攻击者成功入侵了系统的一部分，也无法取得整个系统的访问权限；在造成严重破坏或取得敏感信息之前，他还需要攻破更多的防御措施。

如果多个容器组分属于不同用户，或负责处理敏感数据，它们应该放在虚拟机中运行，并与其他用户的容器或开放了公共接口的容器相隔离。容器开放的端口应该受到严格控制，尤其是对外开放的端口，但即使只对内开放也要注意，以防攻击者从其他已经被成功入侵的容器访问得到。容器可用的资源及功能应仅限于它所需要的，可以通过限制内存使用量、文件访问权限以及内核能力来实现。还可以在内核级别实现更进一步的安全保障，通过运行加固内核，以及使用安全模块，如 AppArmor 或 SELinux。

此外，通过使用监控和审核可以尽早发现攻击。尤其是审核，在容器化系统中进行审核是很特别的，因为我们可以轻松地将容器与创建它的镜像进行对比，从而找出是否有可疑的改动。而且镜像还可以用于离线审查，以确保它运行的软件是最新的和安全的。至于已被入侵的容器，如果它是无状态的话，更换一个新版本就可以了。

就安全性而言，容器发挥了正面作用，因为它能够提供多一重隔离和控制。正确使用容器的系统只会比没有使用容器的系统更安全。

作者简介

Adrian Mouat 是 Container Solutions 公司的首席科学家，该公司专注于在泛欧洲地区提供 Docker 和 Mesos 的相关服务。曾于爱丁堡大学并行计算中心（EPCC）担任应用程序顾问一职。

关于封面

本书封面的动物是一头弓头鲸（学名 Balaena mysticetus）。它是一种体色较深、体型粗壮的鲸，特征是无背鳍。弓头鲸一生都在北极及附近一带的水域渡过，并不会像其他鲸类一样，为了觅食和繁殖而迁徙到低纬度的水域。

弓头鲸体型硕大健壮，长度能达到 16 米（雄性）和 18 米（雌性）。它们有巨大的三角形颅骨，用于凿穿北极冰层呼吸。弓头鲸的下颚明显呈弓形，颜色呈白色，上颚窄，其中容纳的鲸须长度为鲸类中最长（3 米），用于过滤水中的细小猎物。它的一对喷气孔位于头部的最上方，能够把水喷到 6 米高。它的鲸脂厚度堪称鲸类之最，厚达 43~50 厘米。

弓头鲸独来独往，或以 6 头的小群体同行。它们能够在水中逗留最长 1 小时，但通常每次在水中停留的时间为 4~15 分钟。弓头鲸一般每小时能前进 2~5 千米，对鲸来说并不算快，但遇到危险时，它们的速度能达到每小时 10 千米。虽然弓头鲸不太合群，但它们在大型鲸类中最会使用声音沟通。它们在前行、社交和进食时都会在水中使用声音互相沟通。交配季到来时，弓头鲸会发出长而复杂的歌声作为求偶信号。

弓头鲸是已知最长寿的哺乳类动物，寿命可超过 200 岁。2007 年，一头 15 米长的弓头鲸在阿拉斯加海岸被捕获，在它颈部的鲸脂中发现了一块捕鲸枪的残片。该捕鲸枪被追溯至位于马萨诸塞州新贝德福德（New Bedford）的一所主要的捕鲸中心，其生产年份鉴定为 1890 年。其他弓头鲸的年龄则被鉴定为 135~172 岁。弓头鲸曾一度濒临灭绝，在商业捕鲸停止后，数量有所回升。虽然阿拉斯加的原住民为了生计，仍会捕猎少量（25~40 头）的弓头鲸，但预计并不会对它们数量的恢复产生影响。

O’Reilly 出版的图书，封面上很多动物都濒临灭绝。这些动物都是地球的至宝。如果你想知道如何保护这些动物，请访问 animals.oreilly.com。

封面图片出自 *Braukhaus Lexicon*。

O'REILLY®

Docker开发指南

Docker容器给软件的开发、发布和运行提供了简单、快速和可靠的方法，尤其是在动态和分布式的环境中。通过这本实战指南，你将学习到为什么容器如此重要，Docker能带来哪些好处，以及怎样把它变成开发流程的一部分。

本书适合软件开发者、运维工程师和系统管理员，尤其适合对DevOps模式感兴趣的读者。作者将带领你从基础知识出发，直到了解如何在多主机系统上运行数十个拥有联网和调度能力的容器系统，重在让你掌握使用Docker来开发、测试以及部署Web应用。

- 从构建和部署简单Web应用开始了解Docker
- 使用持续部署技术，把应用一天多次推送到生产环境
- 学习各种不同的选项和技术，实现多容器的日志记录和监控
- 剖析联网和服务发现：容器之间如何寻找对方，以及怎样把它们连接起来
- 通过运用容器的编排和集群功能，解决负载均衡、扩展、故障切换以及调度的问题
- 遵守纵深防御和最小权限的原则，确保系统安全
- 利用容器构建微服务架构

Adrian Mouat，Container Solutions公司首席科学家。参与过很多软件项目，既有小型的Web应用，也有大型数据分析软件。

“《Docker开发指南》详尽、实用，尤为难能可贵的是，书中介绍了怎样在Docker的生态系统中把容器化的微服务从开发/测试环境迁移到生产环境。”

——Adrian Cockcroft

Battery Ventures技术分析师

“《Docker开发指南》对Docker和容器生态进行了深入而全面的介绍。这本书注重实践，包含大量范例，因此把其中的概念和技巧运用到实际项目中将非常容易。”

——Pini Reznik

Container Solutions CTO

SYSTEM ADMINISTRATION

封面设计：Randy Comer 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／软件开发

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-44957-3

ISBN 978-7-115-44957-3
定价：79.00元